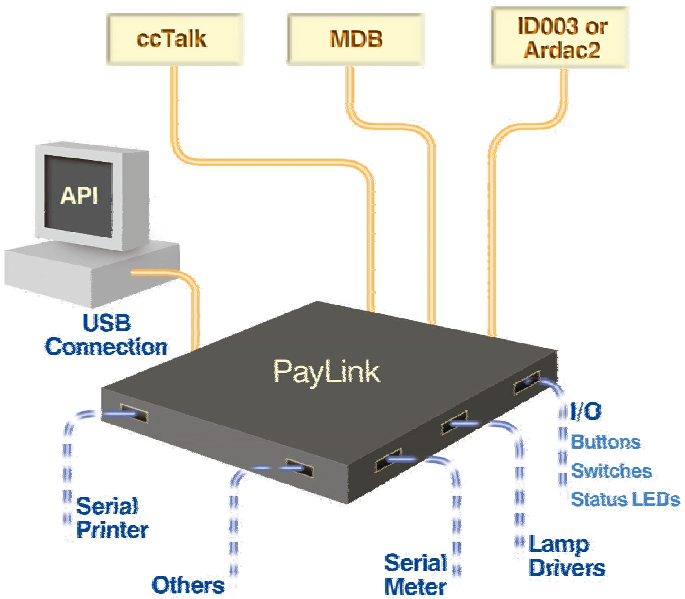
Money Controls

PayLink
Technical Manual



Contents

Figures

Tables

1. Diary of Changes

Issue 1.0……August 2005

- 1st Issue

Issue 1.1…… .November 2005

- Changed the value for cctalk hopper address 10, from 500 to 1
- Corrected a mistake with the pinout for RS232 printer interface
- Change ‘red and black’ to ‘orange and black’ for 24V
- Included information on hotswapping
- Above mentioned changes in line with firmware release 4.1.9.6

Issue 1.2……December 2005

- Corrected a mistake with the cctalk connector pinout information.

2. Overview

2.1 Introduction

PayLink is a simple, compact system that offers trouble free interfacing between a PC and money handling Equipment. **PayLink** allows the rapid integration of a variety of payment peripherals into new machine platforms, without the need for bespoke software.

Designed for use in a wide range of applications

- **AWP**
- **Amusement**
- **Leisure**
- **Change Machines**

Interfaces/protocols supported

- **ccTalk**
- **ID003**
- **MDB**
- **Ardac 2**
- **RS232 serial**

Products supported

- **SR3**
- **Condor Plus**
- **SR5**
- **SR5i**
- **Lumina**
- **Serial Compact Hopper MK2 (SCH2)**
- **Serial Universal Hopper (SUH)**
- **Ardac 5**
- **Serial ticket printer**

I/O supported

- **Up to 16 lamps/low power lamps/relays or switches**
- **Serial electronic meter**

2.2 Contents

PayLink does not come with any cables or software. In order to obtain the software CD (drivers, API) please contact your local Money Controls Technical Services Dept.

Technical Services link: http://www.moneycontrols.com/support/technical_support.asp

Paylink part number: **APCUSBXX00001**

However, Money Controls can provide a development kit, which consists of example cables and a software CD, but this is only available as a 1 off order. Please contact your local Customer Services Dept to place an order.

Customer Services link: http://www.moneycontrols.com/support/customer_support.asp

Paylink development kit part number: **APCUSBXX00002**

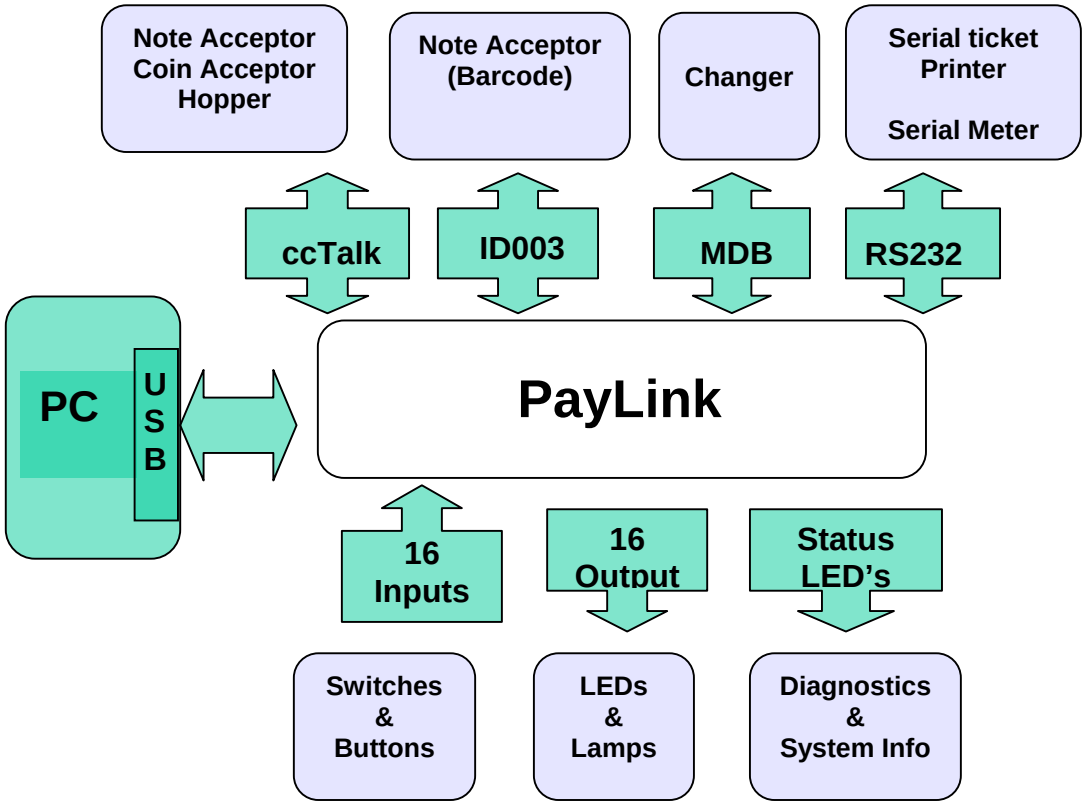
Money Controls recommend purchasing a development kit, in order to aid the integration process in the host machine.
The contents of the **PayLink** Development Kit are as follows:

- **PayLink**
- 1 X cctalk multidrop cable
- 2 X SR5/Lumina cable
- 1 X SR3/Condor Plus cable
- 1 X SCH2 cable – set to address 4
- 1 X SUH cable – set to address 3
- 1 X Serial ticket printer cable
- 1 X Serial meter cable
- 1 X Paylink power cable
- 4 X 20-way headers – for use with inputs/outputs
- 1 X USB Type A – Type B cable
- 1 X Ardac 5 Power cable
- 1 X RJ45-RS232 adapter
- 1 X RJ45 cable

3. Specification

3.1 Functional block diagram

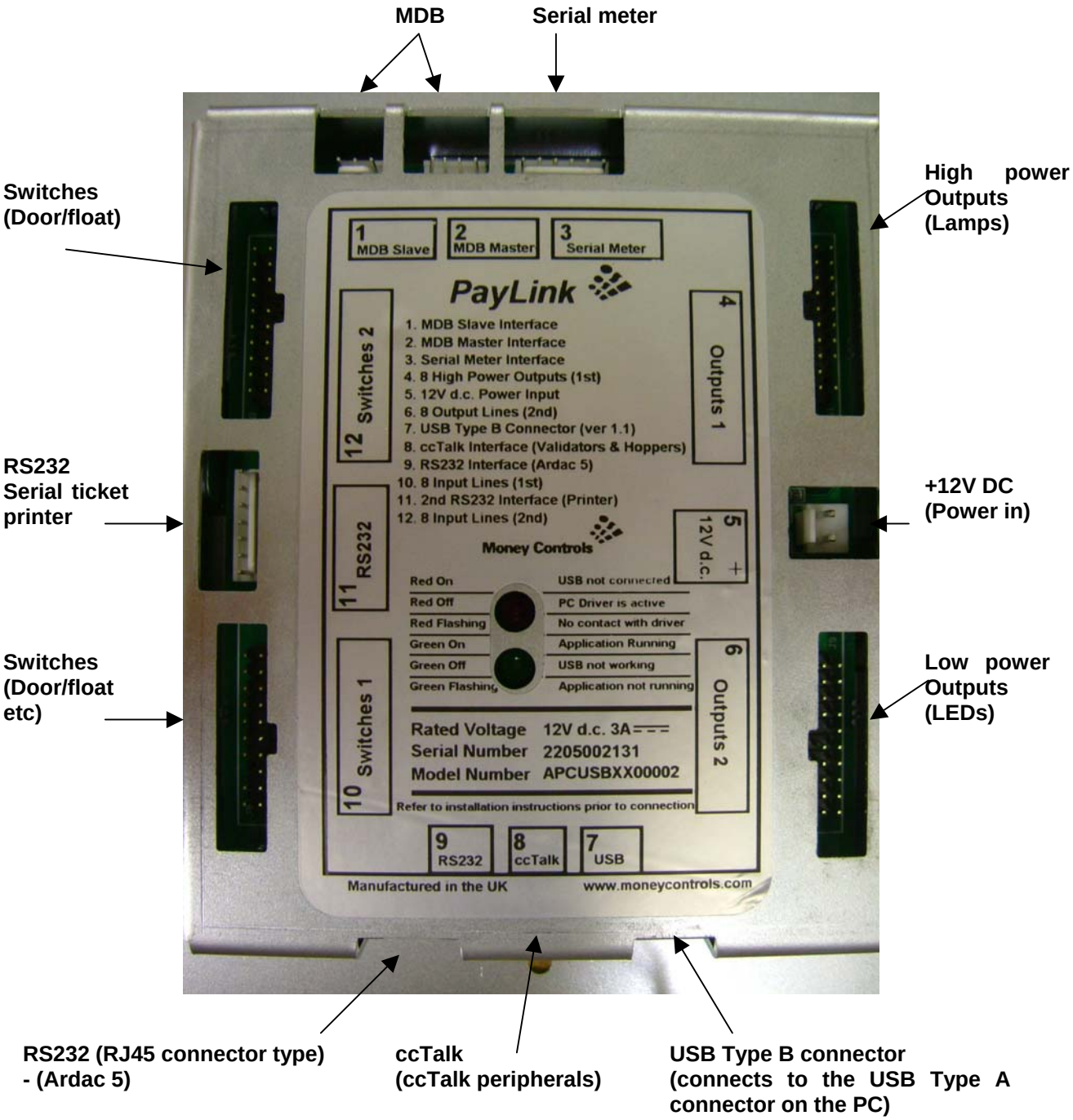
Figure 1: Functional block diagram



3.2 Connector Overview

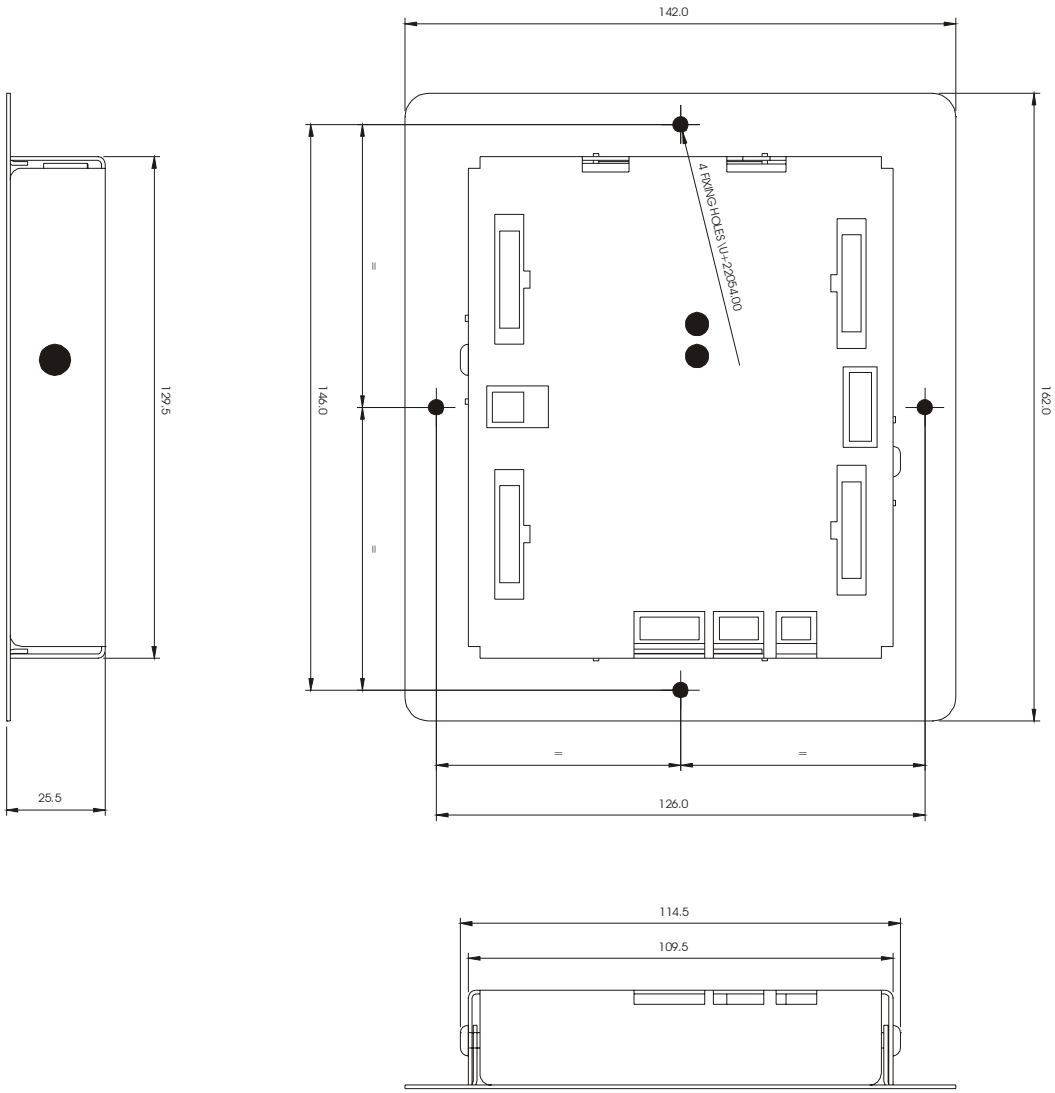
Below is an overview of each connector on **PayLink.**

Figure 2: Connector overview with examples



3.3 Mechanical Dimensions

Figure 3: PayLink mechanical dimensions



3.4 Electrical Specification

Table 1: Electrical Specification

Environmental	
Operating temperature range	0°C to 55°C
Storage temperature range	-20°C to 70°C
Humidity range	Up to 75% RH non-condensing
Electrical - General	
Voltage range	+10.8Vdc to +13.2Vdc (nominal +12Vdc)
Outputs (fuse protected) +12Vdc	2.5A continuous, 5A peak for 200ms
Outputs (fuse protected) +24Vdc	2.5A continuous, 5A peak for 200ms
Electrical – I/O Ports	
16 inputs	Switch inputs 3V3 CMOS thresholds with 3V3 pull-ups, 5mA max.
8 high power outputs	Open drain up to 300mA, max output 36V. (Inductive or resistive)
8 low power outputs	Open drain up to 30mA, max output 12V (resistive only)
Communications Interface	USB Type B interface, V1.1 and above
Protocols support	ccTalk, Ardac 2, ID003, MDB, RS232

4. Installation

4.1 Hardware installation

PayLink connects to the PC via the USB Type A – Type B cable, during the installation process; the LED indicates the current status of **PayLink**.

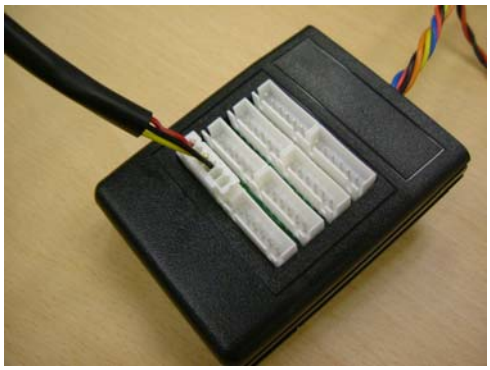
Table 2: Status LED table

RED on	USB not connected (electrical)
RED off	PC driver is active
RED flashing	No contact with PC driver program
GREEN off	USB not working
GREEN flashing	Application not running
GREEN on	Application running

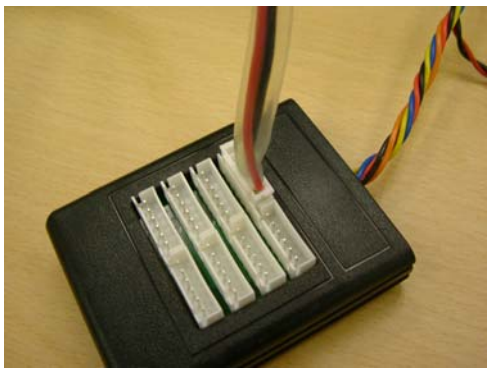
Connect the ccTalk multi drop cable to **Paylink**

Please note: Only one cctalk coin acceptor is supported at once!

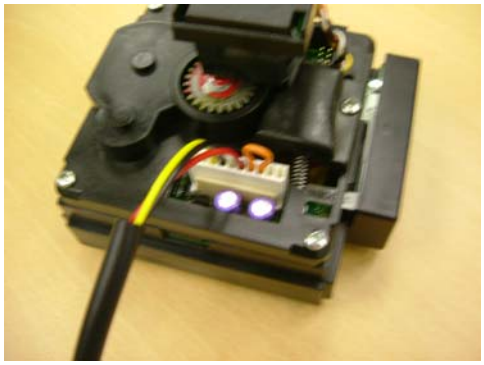
Connect the SR5 cable to the ccTalk multidrop cable and SR5.



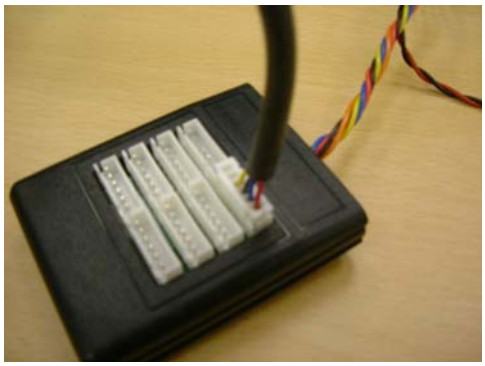
Connect the SR3/Condor Plus cable to the ccTalk multidrop cable and SR3/Condor Plus.



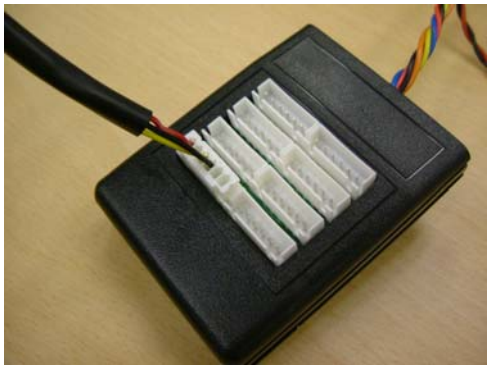
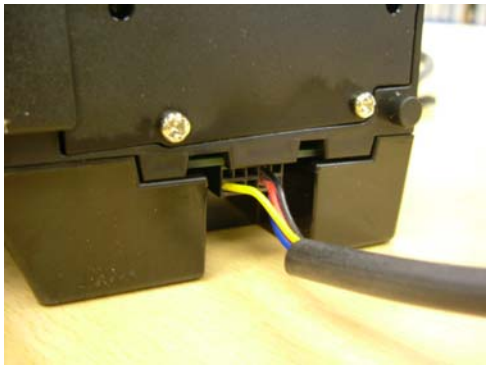
Connect the SCH2 cable to the ccTalk multidrop cable and SCH2.

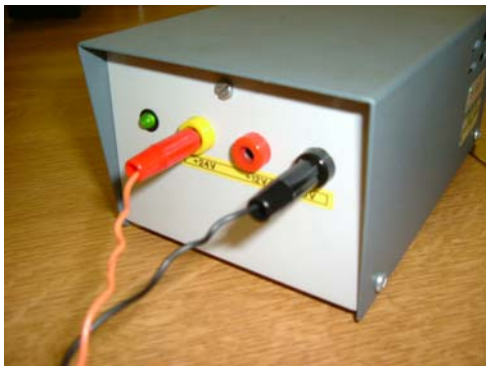
Connect the SUH cable to the ccTalk multidrop cable and SUH.



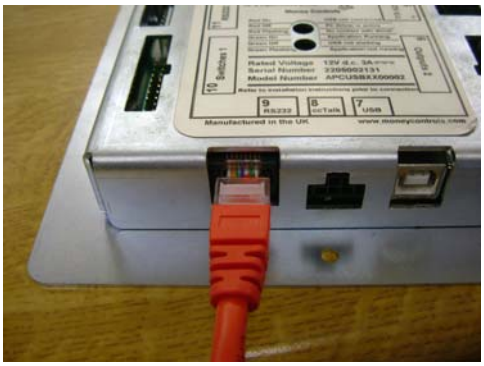
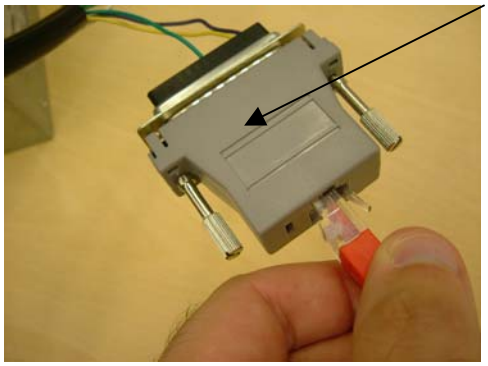

Connect the Lumina cable to the ccTalk multidrop cable and Lumina.

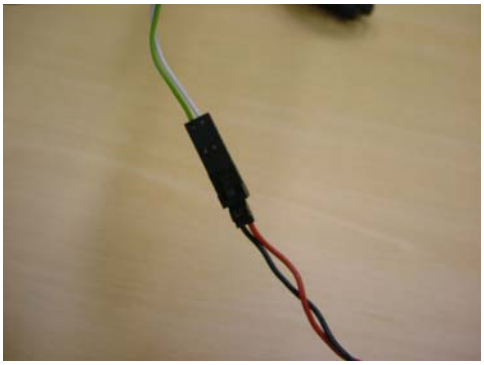
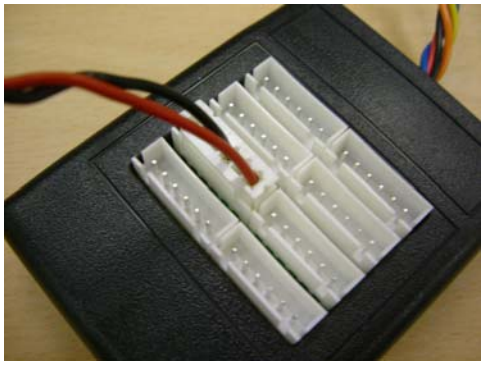
Connect the ccTalk multidrop cable (orange and black) to a +24V dc power supply



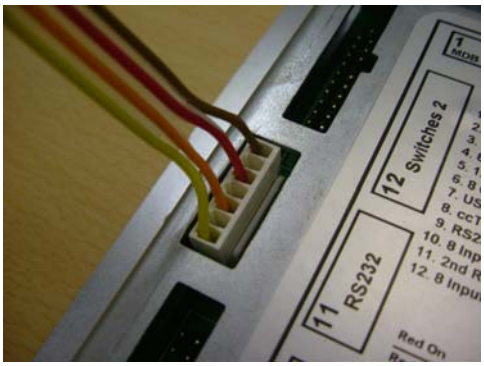
Connect the RJ45 cable to the **PayLink** and Ardac 5 (via the RJ45-RS232 adapter).

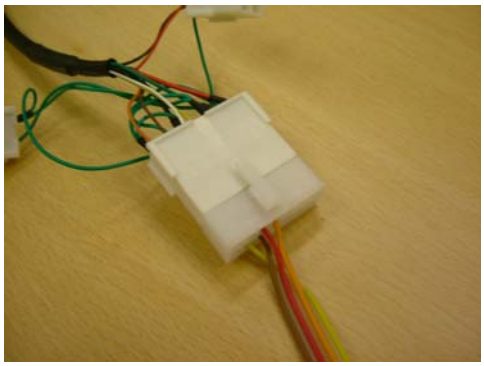



Connect the Ardac 5 power cable to the Ardac 5 and to the multi drop cable

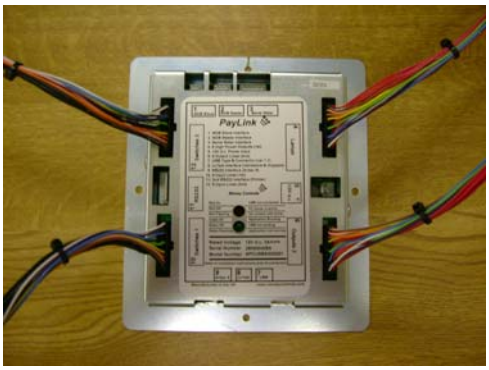
Connect the Serial ticket printer cable to **PayLink** and Serial ticket printer.



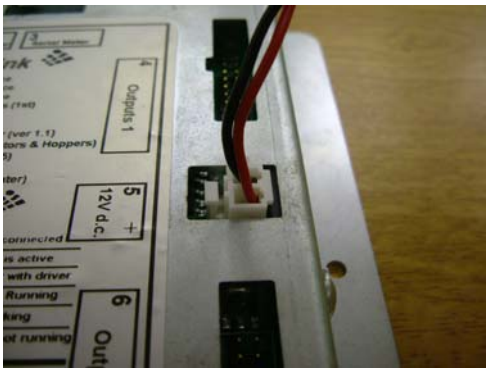


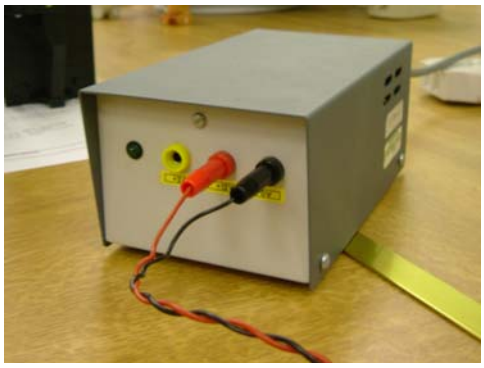
Connect the Serial meter cable to **PayLink** and Serial Meter.

Connect the 4 X 20-way headers to the I/O connectors. *Note: Each 20 way header has a different 'key way' to correspond with the missing pin on the 20-way connectors. The ends of the cables are left open to use as desired.*



Connect **PayLink** to the 2-pin power cable and to a +12V dc power supply. The status LED will show **RED ON**.





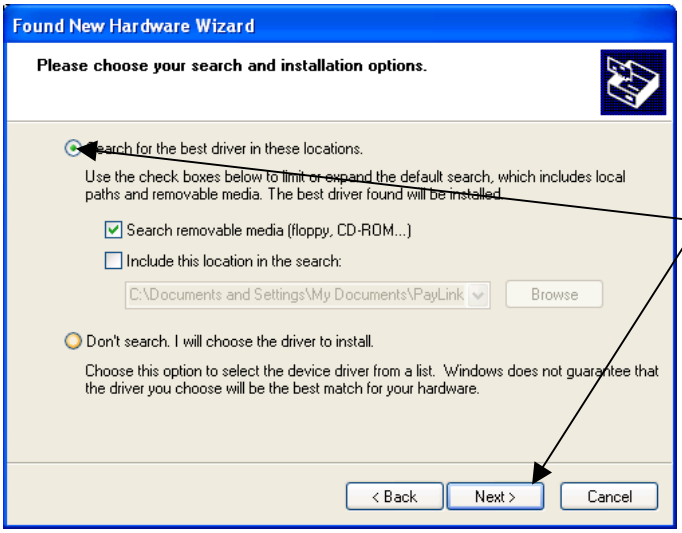
Connect the USB cable to **PayLink** and to the PC.

Windows will indicate that a new USB device has been detected and will prompt for the drivers. The following screen will be shown (this begins the software installation).

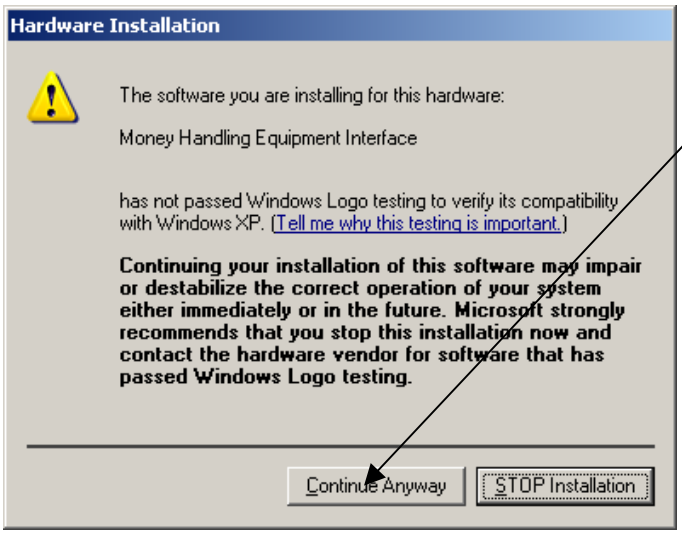
4.2 Software Installation

Note: These instructions are for Windows XP only. Please contact Money Controls for information on installing the software under different operating systems.

Choose **Install from a specific location**, then click **Next**



Choose **Search for the best driver in these locations** then click **Next**

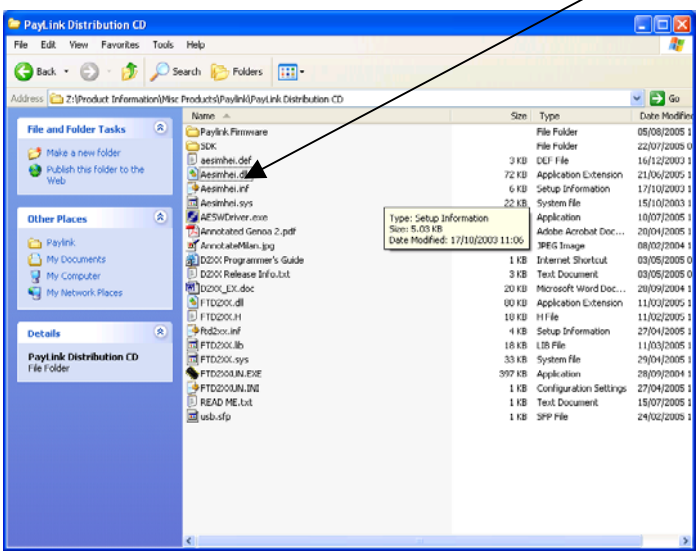


If this screen appears, click **Continue Anyway**

Click **Finish** to complete the software installation for **Paylink**.

To complete the software installation. Take the following step:

In the **PayLink** Distribution CD there is a file called *Aesimhei.dll* – copy this to the **C:\Windows\System32** directory.



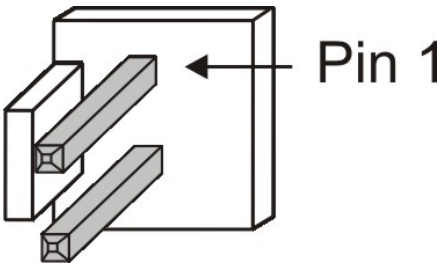
Please note: If this step is not performed, **PayLink** will not function correctly.

Note: At this point, in order to test PayLink. Refer to section 7 ***Using PayLink***

5. Interface

5.1 Power interface

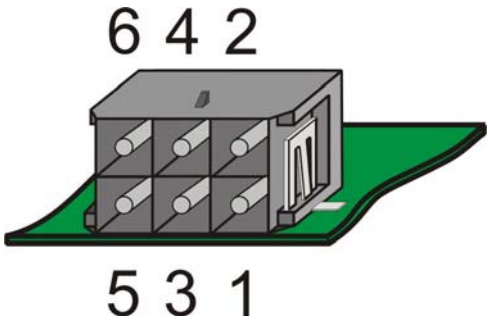
Figure 4: PayLink power interface



Pin	Function
1	GND
2	+12V dc

5.2 ccTalk interface

Figure 5: PayLink cctalk interface

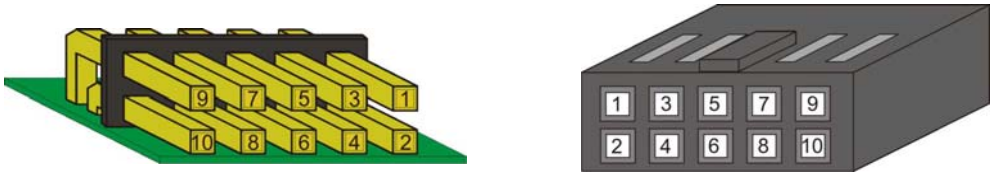


Pin	Function	Pin	Function
1	ccTalk data line	4	+24V Out
2	+12V Out	5	0V In
3	Serial Select / 0V	6	+24V In

IMPORTANT INFORMATION

- **+12V Out is the supply which is provided to PayLink on the 2 pin connector via a polyfuse for protection.**
- **+24V In must be provided by the host machine (in the PayLink development kit, this is shown by orange and black power cables) and is passed through a polyfuse for protection, this becomes +24V Out.**
- **Under no circumstances can any more than 2.5A drawn through the card.**
- **Under no circumstances should PayLink be 'hotswapped'**

Figure 6: Lumina / SR5 ccTalk interface



Pin	Description	Pin	Description	
1	ccTalk data line		**Lumina**	**SR5**
2,3,4,5,6	Not Used			
7	12V	**9**	Not Used	cctalk select line
8	0V	**10**	Not Used	

Figure 7: SR3/Condor Plus ccTalk interface

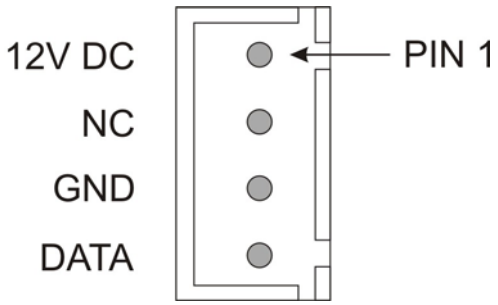
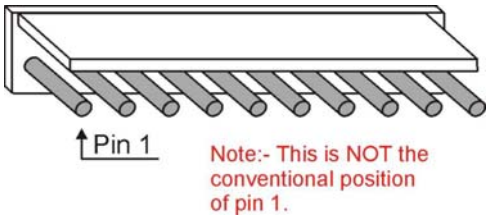


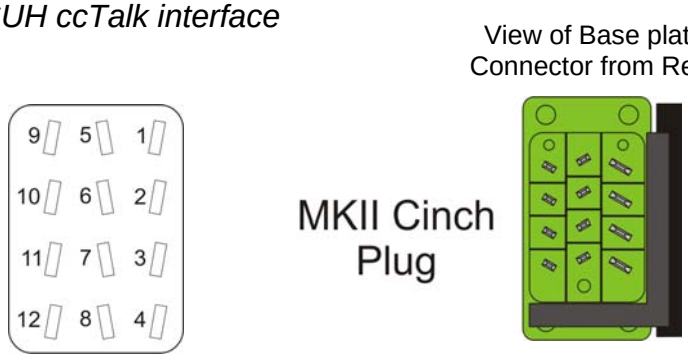
Figure 8: SCH2 ccTalk interface



Pin	Function	Pin	Function
1	Address select 3 - MSB	6,7	0V
2	Address select 2	8	ccTalk data line
3	Address select 1 - LSB	9	N/C
4,5	+Vs	10	/RESET

The address selection process is covered in section 6.3 SCH2/SUH

Figure 9: SUH ccTalk interface

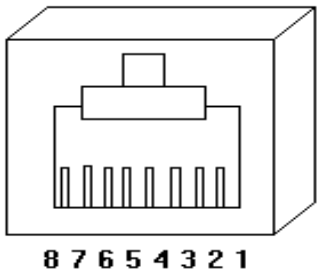


Pin	Function	Pin	Function
1	0V	8	Address Select 2
2,3	N.C.	9	+Vs
4	Address Select 1 - LSB	10,11	N.C.
5	ccTalk data line	12	Address Select 3 - MSB
6,7	N.C.		

The address selection process is covered in section 6.3 SCH2/SUH

5.3 ID003/Ardac 2 interface

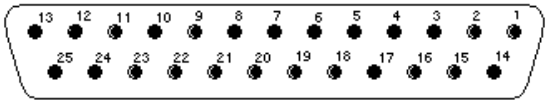
Figure 10: PayLink - ID003/Ardac 2 interface



Pin (Paylink)	Function
3	Rx (Green/White)
4	TX (Blue)
2	GND (Orange)

Figure 11: Ardac 5 - ID003/Ardac 2 interface

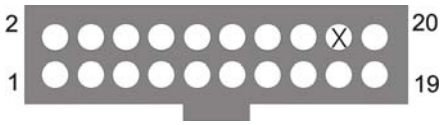
Pin (Ardac5)	Function
2	Rx (Violet)
3	TX (Yellow)
7	GND (Green)



Ardac 5 25 Way D-type (Female) Connector
Important: This view is from the mating side

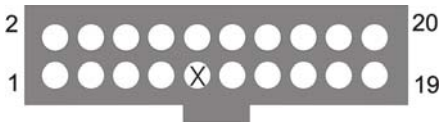
5.4 Auxiliary input/output interface

Figure 12: Connector 4 – High power outputs



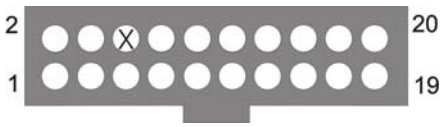
+12V	+12V	N/C	+12V	+12V	+12V	+12V	+12V	Key	+12V
0	1	2	3	N/C	4	N/C	5	6	7

Figure 13: Connector 6 – Low power outputs



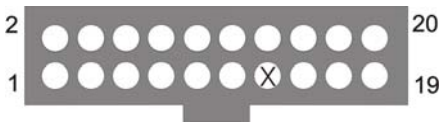
+12V	+12V	N/C	+12V	+12V	+12V	+12V	+12V	N/C	+12V
8	9	10	11	Key	12	N/C	13	14	15

Figure 14: Connector 10 - Switches



0V	0V	N/C	0V	0V	0V	0V	0V	N/C	0V
0	1	2	3	Key	4	N/C	5	6	7

Figure 15: Connector 12 - Switches



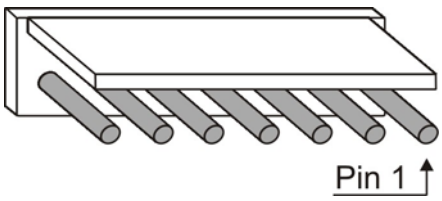
0V	0V	N/C	0V	0V	0V	0V	0V	N/C	0V
8	9	10	11	Key	12	N/C	13	14	15

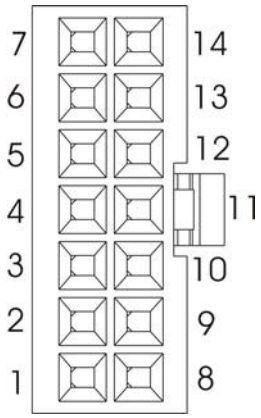
Table 3: I/O Interface

Pin	Conn 4	Conn 6	Conn 10	Conn 12
1	Output 0	Output 8	Input 0	Input 8
2	+12V	+12V	0V	0V
3	Output 1	Output 9	Input 1	Input 9
4	+12V	+12V	0V	0V
5	Output 2	Output 10	Input 2	Input 10
6	N/C	N/C	KEYWAY	N/C
7	Output 3	Output 11	Input 3	Input 11
8	+12V	+12V	0V	0V
9	N/C	KEYWAY	N/C	N/C
10	+12V	+12V	0V	0V
11	Output 4	Output 12	Input 4	Input 12
12	+12V	+12V	0V	0V
13	N/C	N/C	N/C	KEYWAY
14	+12V	+12V	0V	0V
15	Output 5	Output 13	Input 5	Input 13
16	+12V	+12V	0V	0V
17	Output 6	Output 14	Input 6	Input 14
18	KEYWAY	N/C	N/C	N/C
19	Output 7	Output 15	Input 7	Input 15
20	+12V	+12V	0V	0V

5.5 Serial printer interface

Figure 16: PayLink – RS232 Serial Printer Interface

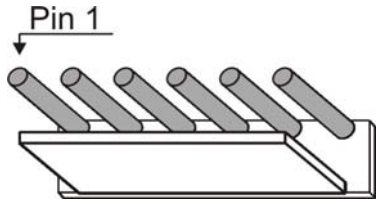




Pin - Paylink	Function	Pin – Serial Printer
1	+24V DC	5
3	TxD (from PayLink)	11
5	RxD (to PayLink)	12
7	GND	6

5.6 Serial meter interface

Figure 17: PayLink serial meter interface



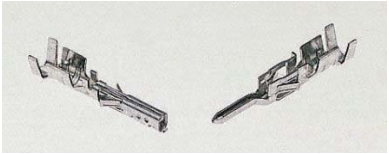
This is a 1 to 1 connection between **PayLink** and the Serial meter.

Pin (Meter)	Function	Pin (Meter)	Function
1	SPI Data Output	4	SEC Reset
2	SPI Clock Input	5	+12V Supply
3	SPI Data Input	6	0V Supply

5.7 Connector details

Below is information of some recommended connector and crimp types. These are only recommendations and may not be available in all countries of the world.

Table 4: Connector details

	Connector	Crimp
PayLink ccTalk		
RS No.	**233-2769**	**233-3009**
SR3 ccTalk		
JST Part No.	**XHP-4**	**SXH-001T-P0.6**
ccTalk SR5 / Lumina (10 way)		
RS No.	**360-6229 (10)** **360-6207 (16)**	**360-6869**
Serial Compact Hopper MK2		
RS No.	**296-5022**	**467-598**
Serial Universal Hopper		**Not applicable**
RS No.	**466-078**	**Not applicable**
Ardac 5		**Not applicable**
Farnell UK No.	**225186**	**Not applicable**

PayLink Power		**Not available**
Leotronics Part no	**3950-2021**	**3953-2000**

6. Peripheral Features/Support

6.1 SR3/Condor Plus/SR5/SR5i

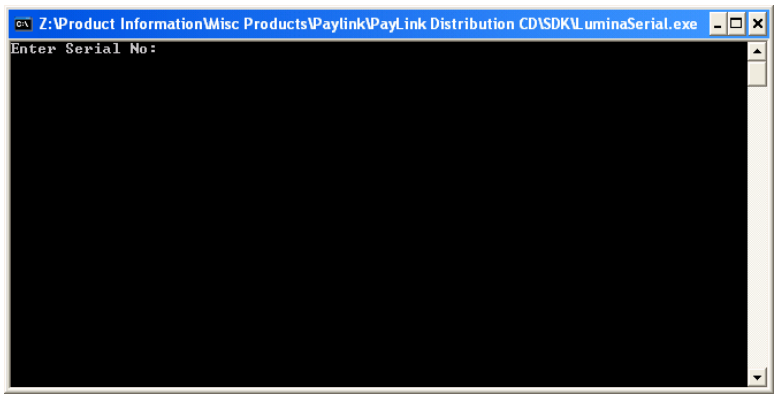
- At present, only one coin acceptor, at address 2, is supported.
- A complex system of routing is provided, which supports the diversion of coins.
- Both individual coins and the entire unit can be easily inhibited.
- The automatic retrieval from the unit of the value of each coin is supported.

6.2 Lumina

- At present, only one note acceptor, at address 40, is supported.
- **PayLink** fully supports the ccTalk encryption scheme needed to communicate with Lumina.
- Both individual notes and the entire unit can be easily inhibited.
- The automatic retrieval from the unit of the value of each note is supported.
- The default Lumina 6-digit security code is 123456. To use a Lumina with a different security code an application is provided. Luminaserial.exe is found in the following directory

\PayLink Distribution CD\SDK

Run LuminaSerial.exe – the following screen will be shown:



Enter the Lumina 6-digit security code (found on a label on the top of Lumina) and click **Enter**. This will close the application. **PayLink** will now work with the code specified. To change to a different code, run LuminaSerial.exe again to change the code.

6.3 SCH2/SUH

➢ Currently, 8 Hoppers, at addresses 3 to 10, are supported and the pre-set values are linked to the cctalk address (shown below).

Table 5: Hopper address – coin value

Address	Coin Value	Address	Coin Value
3	100	**7**	10
4	50	**8**	5
5	25	**9**	200
6	20	**10**	1

➢ The hopper addresses is selected by hardwiring the connector (shown below)

Table 6: Hopper address wiring

X = Connect to +Vs (Pins 4, 5)			ccTalk Address
Address select 3	Address select 2	Address select 1	
			3
		X	**4**
	X		**5**
	X	**X**	**6**
X			**7**
X		**X**	**8**
X	**X**		**9**
X	**X**	**X**	**10**

➢ It is recommend to use only use 24V hoppers.
➢ 12V SCH2 hoppers can be used, but you must not power via **PayLink**, as the current consumption will be too high. Under no circumstances can any more than 2.5A drawn through the card.
➢ Hopper level sense is currently not supported.
➢ The below hoppers values have been implemented from PayLink firmware version 4-1-9-6 and above.

6.4 Ardac 5

- Paylink supports either ID003 or Ardac 2 protocol but not both. In order to convert from Ardac 2 protocol to the ID003 protocol (and vice versa), the necessary firmware needs to be programmed into Paylink. Refer to section 7.4 Upgrading PayLink firmware for information on how to do this.
- Must be powered at 24V as the current consumption at 12V will be too high. Under no circumstances can any more than 2.5A drawn through the card.
- Both individual notes and the entire unit can be easily inhibited.
- The automatic retrieval from the unit of the value of each note is supported.

6.5 Serial ticket printer

- The printer needs to be preloaded with a template.
- Currently only supports Futurelogic GEN2 ticket printer. Please contact Money Controls Technical Services for details.

6.6 MDB

- At the time of writing there is currently no **PayLink** software for the MDB interface. Please contact Money Controls for further information.

6.7 Inputs

- 16 Individual external switches are supported by the unit, and are easily accessible by the user's application.
- Provision is made for the user's application to easily use switches in two modes:

1. Key Press - Where a button may be pressed several times and it is important to know how many times
2. State - Where the switch changes over a long time frame and all the application needs to know is where the switch is at any instant.

6.8 Outputs

- 8 Individual external LED's are supported by the unit, and are easily accessible by the user's application.
- 8 high power (lamp) outputs are supported by the unit, and are easily accessible by the user's application.

6.9 Serial meter

- One external meter with an SPI interface corresponding to that defined by Starpoint is supported.
- The **PayLink** board fully supports all 31 of the Starpoint's counters.
- Provision is made to allow the user's application to easily support the BACTA standard for displaying counter values, as well as to implement any other scheme.
- The **PayLink** board continually checks that the meter is operation.

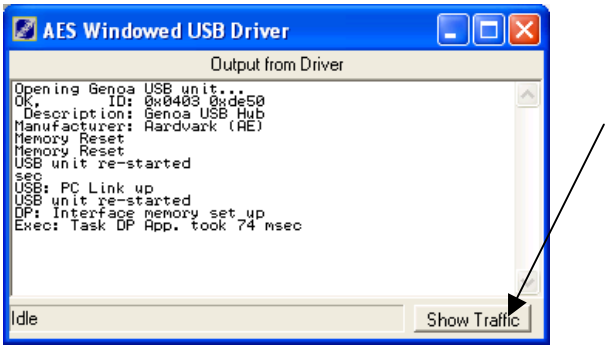
7. Using PayLink

This section shows how to run and use various programs, all of which are provided on the **PayLink** distribution CD.

- **AESWDriver** (the Paylink driver)
- **Diag** (diagnostics program)
- **Demo** (API example)
- **Firmware** upgrade program

7.1 AESW Driver

AESWDriver.exe is found in the **PayLink Distribution CD** directory. When the application is run, the following screen will be shown.



Clicking the **Show Traffic** button will show all the comms between PC and Paylink.

The contents of this screen should be similar to the one shown above. The status LED on **PayLink** will now turn **GREEN** to indicate that the driver is working correctly.

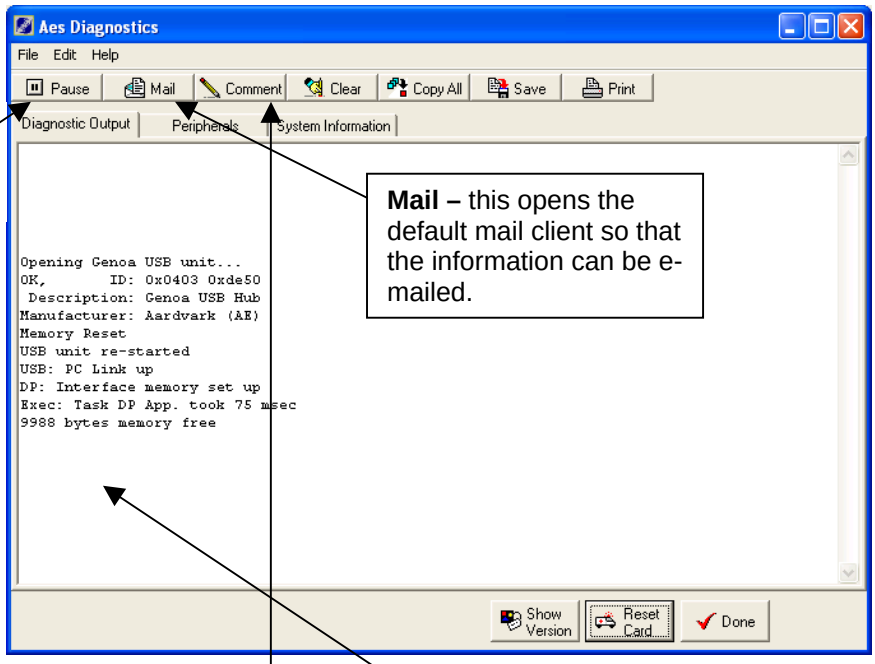
Refer to Table 2: Status LED table for information.

This driver MUST be run before running the demo software.

7.2 Diag.exe

This is a Diagnostics program, which shows various information about **PayLink**, such as the peripherals, which are connected, the version number of Paylink firmware. Diag.exe is found in the following directory: **PayLink Distribution CD\SDK** When the application is run, the following screen will be shown:

Pause – This pauses the application.



Mail – this opens the default mail client so that the information can be e-mailed.

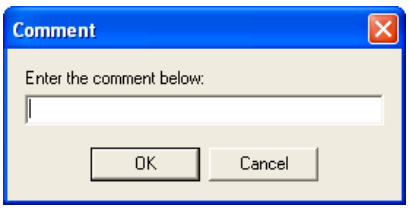
Clear – clears the screen.

Copy all – this copies the shown text to clipboard.

Save – This saves the text in a log file.

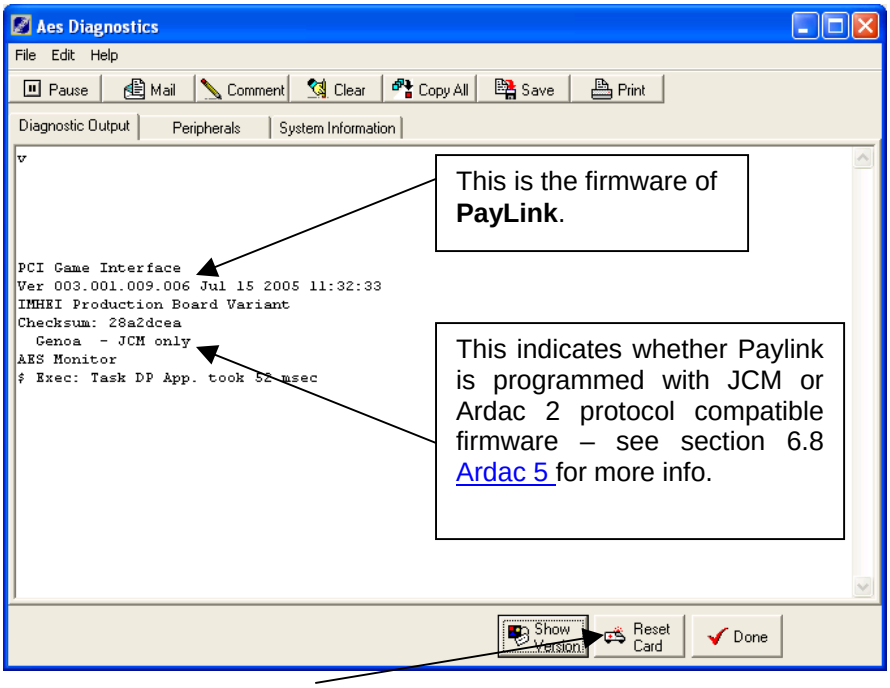
Print – This prints the current text.

Clicking the **Comment** button, allows a comment to be added, the following screen will appear.

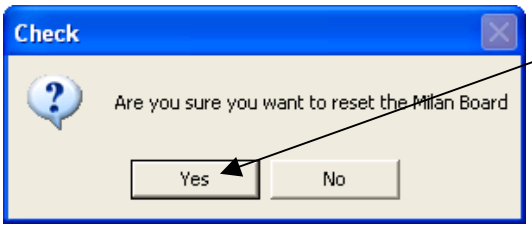


A comment will then appear in the diagnostics window.

Clicking on the **Show Version** button will show the following screen.

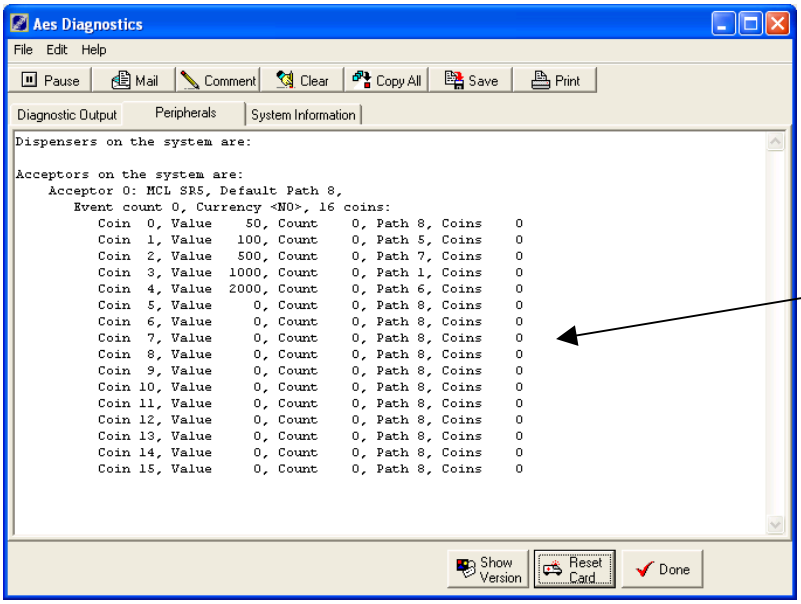


Click on the **Reset Card** button will show the following screen.



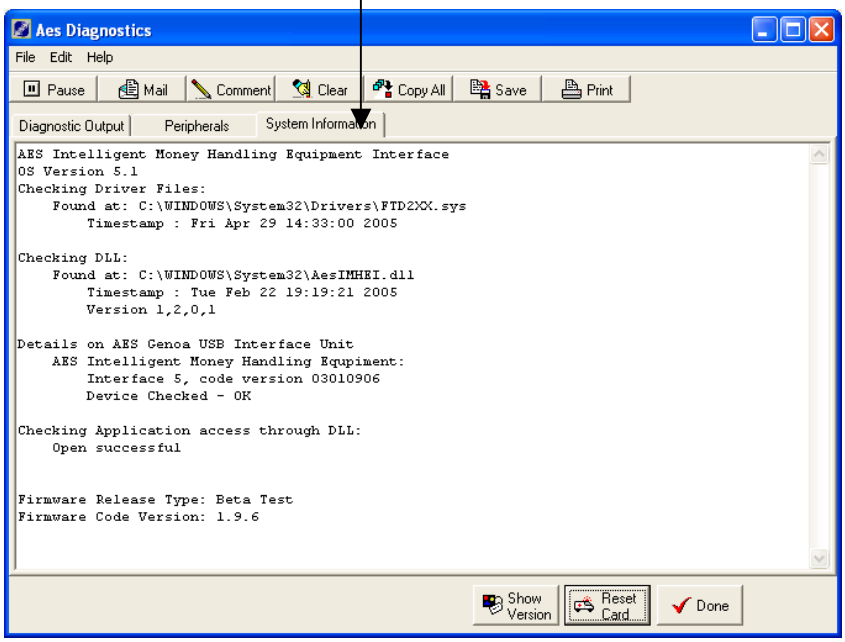
Choose **Yes** to reset **PayLink**. The following screen will be shown.

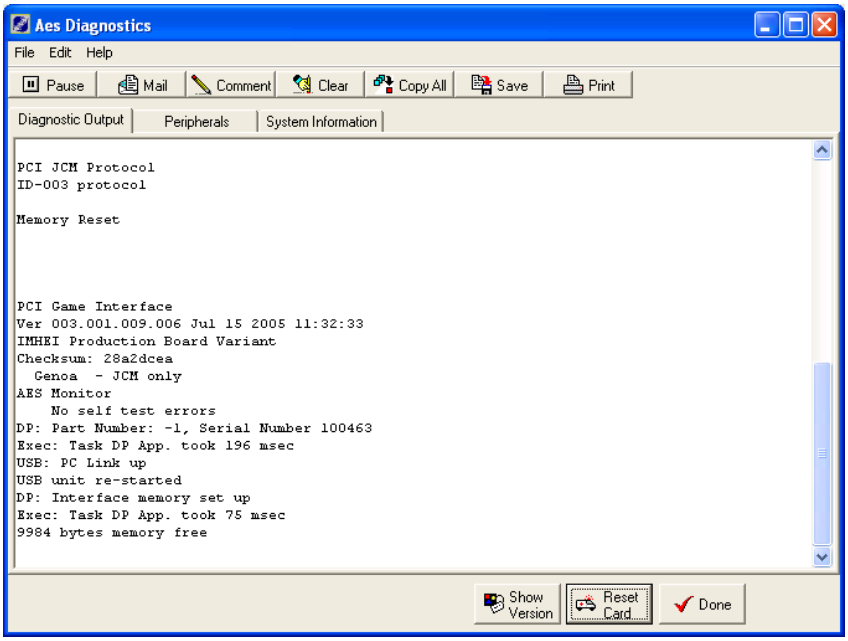
Click on the **Peripherals** tab to see which peripherals are connected. The below screen shows an example.



This example shows an SR5 coin acceptor, and information about the coin paths and values etc.

Click on the **System Information** tab to display various system information about **PayLink**. An example is shown below.





Click **Done** to close the Diagnostics application.

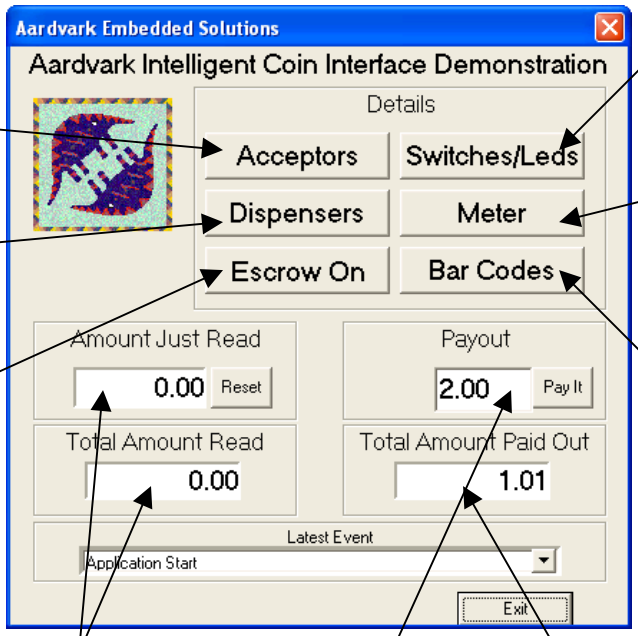
7.3 Demo.exe

This is a free of charge API example, which also doubles up as a quick and easy way to test/demo **PayLink** before the software writing can begin. The application is called Demo.exe and is in the following location: **PayLink Distribution CD\SDK**

Click **Acceptors** to show the current connected Coin/Note acceptors

Click **Dispensers** to show the current connected Hoppers

Click **Escrow** for the Escrow control



Click **Switches/LEDs** to control the Inputs/outputs

Click **Meter** to show the current connected Serial Meter

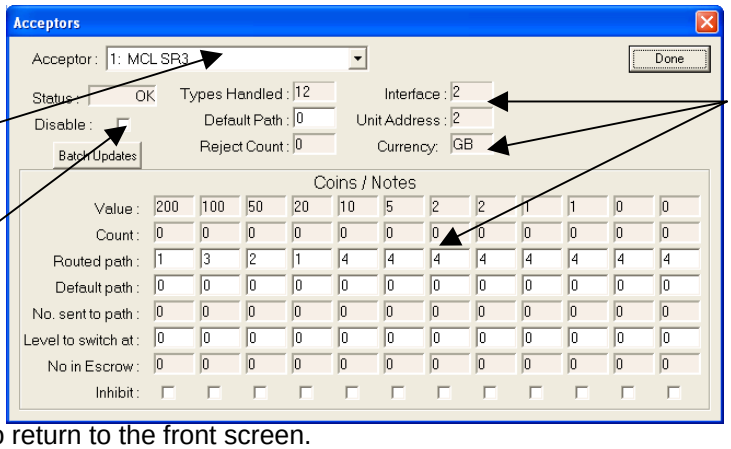
Click **Barcodes** to control the barcode features (Ardac 5/Serial ticket printer)

In this screen coins and notes CAN be entered into the peripherals – they will be displayed in the **Amount Just Read** box. This will reset to 0 after a fixed time. The **Total Amount Read** box will display the cumulative total.

The **Payout** box shows the value to be paid out. Click the **Pay It** button to pay out the desired value. **Paylink** will decide how to pay out the value depending on which value hoppers are connected. The **Total Amount Paid Out** shows the cumulative total.

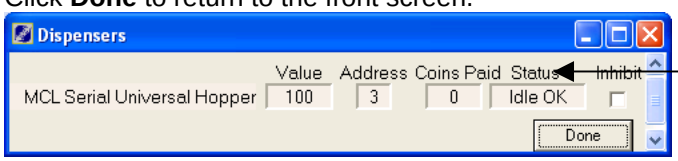
This drop down menu shows the acceptors connected

Click **Disable** to disable the acceptor selected

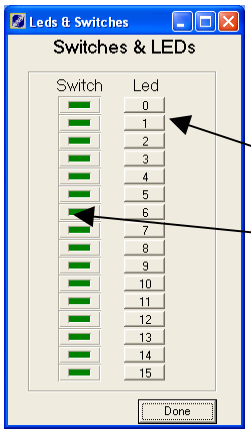


Various information about the selected acceptor such as currency, coins programmed etc

Click **Done** to return to the front screen.



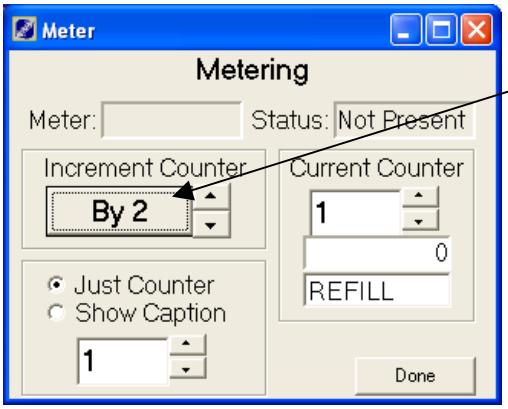
Click on the **Dispensers** button and this screen will be shown. Various information about the connected **Dispensers** is shown.



Click on the Switches/LEDs button to see the following screen.

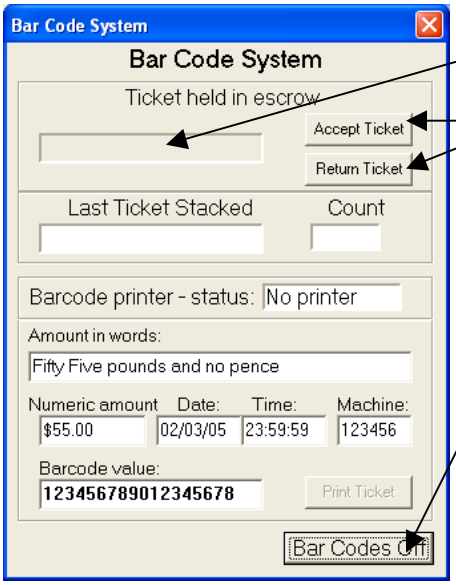
Click on the Led buttons to drive the LED output.

The switch box will light when the switch inputs are activated.



Click on the **Meter** button to show this screen. The counter can be incremented using the **Increment Counter** button.

Click on the **Barcodes** button to show the following screen.



When a barcode is inserted, the number will be shown here. Click **Accept ticket** or **Return ticket** to proceed.

The barcodes screen can be exited using the **Bar Codes Off** button

7.4 Upgrading PayLink firmware

PayLink has an on board flash device, which can be reprogrammed using a small application provided. The application is found in the following directory (see below)

PayLink Distribution CD\Paylink Firmware\

Choose the firmware file provided, and double click. The below screen will be shown.

Once complete, the **AES Programming Utility** will self terminate. It is recommended to recycle the power to **PayLink** for the new firmware to take effect.

8. API Software Guide – Introduction

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

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

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

Getting Started:
These are the minimum set of "vanilla" functions that may be used to get a working *demonstration* program running.

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

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

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

Full Game System:
These build on the set of functions provided within the "Getting Started" section.

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

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

Engineering Support:
These functions provide full-blown diagnostics and reconfiguration facilities.

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

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

9. API Software Guide – Getting Started

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

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

9.1 OpenMHE

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

long OpenMHE (void);

Parameters
None

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

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

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

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

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.

9.2 EnableInterface

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

void EnableInterface (void) ;

Parameters
None

Return Value
None

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

9.3 DisableInterface

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

void DisableInterface (void) ;

Parameters
None

Return Value
None

Remarks
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.

9.4 CurrentValue

Synopsis
Determine the current monetary value that has been accepted

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

long CurrentValue (void) ;

Parameters
None

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

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

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.

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.

9.5 PayOut

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

void PayOut (long Value) ;

Parameters
Value

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

Return Value
None

Remarks
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.

9.6 PayStatus

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

long LastPayStatus (void) ;

Parameters
None

Return Values

Value	Meaning	Mnemonic
0	The interface is in the process of paying out	PAY_ONGOING
1	The payout process is up to date	PAY_FINISHED
-1	The dispenser is empty	PAY_EMPTY
-2	The dispenser is jammed	PAY_JAMMED
-3	Dispenser non functional	PAY_US
-4	Dispenser shut down due to fraud attempt	PAY_FRAUD
-5	The dispenser is blocked	PAY_FAILED_BLOCKED
-6	No Dispenser matches amount to be paid	PAY_NO_HOPPER
-7	The dispenser is inhibited	PAY_INHIBITED
-8	The internal self checks failed	PAY_SECURITY_FAIL

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

If one out of multiple hoppers has a problem, the **PAYLINK** card will do the best it can. If it can not pay out the entire amount, the status will reflect the last attempt.

9.7 IndicatorOn / IndicatorOff

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

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

Parameters
IndicatorNumber

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

Return Value
None

Remarks
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.

9.8 SwitchOpens / SwitchCloses

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

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

Parameters
SwitchNumber

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

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

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

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

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.

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

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

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).

10. API Software Guide - Getting Started Code Examples

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

10.1 Currency Accept

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

    long OpenStatus = OpenMHE() ;

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

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

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

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

10.2 Currency Payout

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

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

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

10.3 Indicator Example

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

   if (!OpenStatus)
      {
      EnableInterface() ;

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

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

      DisableInterface() ;
      }
```

```
   }
```

10.4 Switch Example

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

   if (!OpenStatus)
      {
      EnableInterface() ;

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

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

      DisableInterface() ;
      }
   }
```

11. API Software Guide - Full game system

11.1 CurrentPaid

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

long CurrentPaid (void) ;

Parameters
None

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

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

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.

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.

11.2 SystemStatus

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

long SystemStatus (void) ;

Parameters
None

Return Value
Zero if all devices are completely normal.

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

Remarks
Although this call is available, it is currently un-implemented.
This returns a logical OR of the status of all of the individual device statuses.

11.3 NextEvent.

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

int NextEvent(EventDetailBlock* EventDetail);

Parameters

EventDetail
NULL, or the address of the structure at which to store the details of the event.

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

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

Driver Software	Raw Code for Event	Raw Code for Fault
cctalk coin		
cctalk note		
Ardac 5 note		
Hi2 coin		
JCM note		
GPT note		

11.4 AvailableValue

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

long AvailableValue (void) ;

Parameters
None

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

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

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

11.5 ValueNeeded

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

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

void ValueNeeded (long Amount) ;

Parameters
Amount

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

Return Value
None

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

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

11.6 ReadAcceptorDetails

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

bool ReadAcceptorDetails (long Number, AcceptorBlock* Snapshot) ;

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

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

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

11.7 WriteAcceptorDetails

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

void WriteAcceptorDetails (long Number, AcceptorBlock* Snapshot) ;

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

Return Value
None.

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

11.8 ReadDispenserDetails

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

bool ReadDispenserDetails (long Number, DispenserBlock* Snapshot) ;

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

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

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

11.9 WriteDispenserDetails

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

void WriteDispenserDetails (long Number, DispenserBlock* Snapshot) ;

Parameters

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

Return Value
None.

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

11.10 SetDeviceKey

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

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

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

Return Value
None

Remarks
At present, this can only be used for a Lumina acceptor at address 40 on interface 2, the cctalk interface. The key (as 6 hex digits) is used as the encryption key.

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

11.11 SerialNumber

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

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

Parameters
None

Return Value
32 bit serial number.

Remarks
A serial number of -1 indicates that a serial number has not been set in the device.

A serial number of 0 indicates that the device firmware does not support serial numbers

11.12 Escrow

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

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

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

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

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

11.13 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

11.14 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
If any currency is currently held in escrow when this call is made, it will be accepted without comment.

11.15 EscrowThroughput

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

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

long EscrowThroughput (void) ;

Parameters
None

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

Remarks
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.

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.

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.

11.16 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

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.
If no currency is currently held in escrow when this call is made, it will be silently ignored.

11.17 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
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.
If no currency is currently held in escrow when this call is made, it will be silently ignored.

11.18 Bar Codes

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

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

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

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

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

11.19 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

11.20 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
If a Barcoded ticket is currently held when this call is made, it will be returned without comment.

11.21 BarcodeInEscrow

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

bool BarcodeInEscrow (char BarcodeString[19]) ;

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

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

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

11.22 BarcodeStacked

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

long BarcodeStacked (char BarcodeString[19]) ;

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

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

Remarks
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.

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.

11.23 BarcodeAccept

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

void BarcodeAccept (void) ;

Parameters
None

Return Value
None

Remarks
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.
If no ticket is currently held when this call is made, it will be silently ignored.

11.24 BarcodeReturn

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

void BarcodeReturn (void) ;

Parameters
None

Return Value
None

Remarks
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.
If no ticket is currently held when this call is made, it will be silently ignored.

11.25 BarcodePrint

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

void BarcodePrint (TicketDescription* TicketContents) ;

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

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

Return Value
None

Remarks
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.
Before issuing this call the application should ensure that **BarcodePrintStatus** has returned a status of **PRINTER_IDLE**
The mechanics of the priniting mechanism rely on **BarcodePrintStatus** being called regularly after this call, in order to “stage” the data to the interface.

11.26 BarcodePrintStatus

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

long BarcodePrintStatus (void) ;

Parameters
None

Return Value

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

Remarks

The mechanics of the priniting 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.

Any reported fault that requires an operator action will cause the **PRINTER_FAULT** bit to be set.

A **PRINTER_NONE** status will be reported if the printer is powered off after having been working.

12. API Software Guide - 'C' Program Structures and Constants

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

12.1 System

```
enum SystemConstants
    {           // This area is still under development
    SYSTEM_MASK           = 0xf0000000,
    INTERFACE_FAILED      = 0x80000000
    DISPENSER_MASK        = 0x0fff0000,
    ACCEPTOR_MASK         = 0x0000ffff
  } ;
```

12.2 AcceptorBlock

Constants for AcceptorBlock

```
enum AcceptorConstants
    {
    ACCEPTOR_DEAD      = 0x00000001,    // No response to communications for this device
    ACCEPTOR_DISABLED  = 0X00000004,    // Disabled by Interface
    ACCEPTOR_INHIBIT   = 0X00000008,    // Specific by Application
    ACCEPTOR_BUSY      = 0x00000020,    // Reported from device
    ACCEPTOR_FAULT     = 0x00000040,    // Reported from device

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

Structures for AcceptorBlocks

```
typedef struct
    {
    long        Value ;                 // Value of this coin
    long        Inhibit ;               // Set by PC: this coin is inhibited
    long        Count ;                 // Total number read "ever"
    long        Path ;                  // Set by PC: this coin's chosen output path
    long        PathCount ;             // Number "ever" sent down the chosen Path
    long        PathSwitchLevel ;       // Set by PC: 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 (usual max 1)
    char        CurrencySet ;           // Currency set to which this coin belongs
    } AcceptorCoin ;

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

12.3 DispenserBlock

Constants for DispenserBlock

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

                // Coin Count Status Values
    DISPENSER_COIN_NONE    = 0,           // No dispenser coin reporting
    DISPENSER_COIN_LOW     = 1,           // Less than the low sensor level
    DISPENSER_COIN_MID     = 2,           // Above low sensor but below high
    DISPENSER_COIN_HIGH    = 3,           // High sensor level reported
                                          // Note - count is set to an approximation
    DISPENSER_ACCURATE     = -1,          // Coin Count reported by Dispenser
    } ;
```

Structure for DispenserBlock

```
typedef struct
    {
    long        Unit ;                  // Specification of this unit
    long        Status ;                // Individual Dispenser status
                                        // This takes the same values as PayStatus()
    long        InterfaceNumber ;       // The bus / connection
    long        UnitAddress ;           // For addressable units
    long        Value ;                 // The value of the coins in this dispensor
    long          Count ;                       // Number dispensed since interface
commissioned
    long        Inhibit ;               // Set to 1 to inhibit Dispenser
    long        Currency ;              // The currency code reported by
                                        // an intelligent dispenser
    long        CoinCount ;             // The number of coins in the dispenser
    long        CoinCountStatus ;       // Flags Relating to Coin Count (See above)
    } DispenserBlock ;
```

12.4 EventDetailBlock

Structure for EventDetailBlock

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

Event Codes for NextEvent / EventDetailBlock

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

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

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

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

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

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

    IMHEI_OVERFLOW                      = 0x1f,


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


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



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

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

12.5 Device Identity Constants

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

Example

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

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

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

```
enum GenericDevices
    {
    DP_GENERIC_MASK               =  0xff000000,

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

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

enum InterfaceNumbers
    {     // These describe the interface via which this device is connected:
    DP_INTERFACE_MASK             =  0x00ff0000,
    DP_INTERFACE_UNIT             =  0x00000000,
    DP_ONBOARD_PARALLEL_INTERFACE =  0x00010000,
    DP_CCTALK_INTERFACE           =  0x00020000,
    DP_SSP_INTERFACE              =  0x00030000,
    DP_HII_INTERFACE              =  0x00040000,
    DP_ARDAC_INTERFACE            =  0x00050000,
    DP_JCM_INTERFACE              =  0x00060000,
    DP_GPT_INTERFACE              =  0x00070000,
    DP_MDB_INTERFACE              =  0x00080000,
    } ;

enum ManufacturerIdentities
    {                              // These describe the manufacturer of the device.
    DP_MANUFACTURER_MASK          =  0x0000ff00,
    DP_MANU_UNKNOWN               =  0x00000000,
    DP_MANU_MONEY_CONTROLS        =  0x00000100,
    DP_MANU_INNOVATIVE_TECH       =  0x00000200,
    DP_MANU_MARS_ELECTRONICS      =  0x00000300,
    DP_MANU_AZKOYEN               =  0x00000400,
    DP_MANU_NRI                   =  0x00000500,
    DP_MANU_ICT                   =  0x00000600,
    DP_MANU_JCM                   =  0x00000700,
    DP_MANU_GPT                   =  0x00000800,
 } ;
```

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

                                                          // Money Controls Devices
    DP_MCL_SUH1                   =  2 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_PAYOUT_DEVICE,

    DP_MCL_SR3                    =  2 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_SR5                    =  3 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_CONDOR                 =  6 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_LUMINA                 =  5 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

    DP_MCL_7200                   =  6 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

    DP_MCL_WACS                   =  1 | DP_MANU_MONEY_CONTROLS
                                       | DP_ARDAC_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

    DP_MCL_VORTEX                 =  1 | DP_MANU_MONEY_CONTROLS
                                       | DP_MDB_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_VORTEX_TUBE            =  1 | DP_MANU_MONEY_CONTROLS
                                       | DP_MDB_INTERFACE
                                       | DP_COIN_PAYOUT_DEVICE,

    DP_MCL_GLOBAL                 =  2 | DP_MANU_MONEY_CONTROLS
                                       | DP_MDB_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_GLOBAL_TUBE            =  2 | DP_MANU_MONEY_CONTROLS
                                       | DP_MDB_INTERFACE
                                       | DP_COIN_PAYOUT_DEVICE,

                                                    // Azkoyen Devices
    DP_MCL_AZK                    =  1 | DP_MANU_AZKOYEN
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_PAYOUT_DEVICE,

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

    DP_MARS_CASHFLOW_9500         =  2 | DP_MANU_MARS_ELECTRONICS
                                       | DP_HII_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

                                                     // Innovative Devices
    DP_INNOV_NV4                  =   4 | DP_MANU_INNOVATIVE_TECH
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

    DP_INNOV_NV7                  =   7 | DP_MANU_INNOVATIVE_TECH
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

    DP_INNOV_NV8                  =   8 | DP_MANU_INNOVATIVE_TECH
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,
```

```
                                        // NRI Devices
    DP_NRI_G40                  =  1 | DP_MANU_NRI
                                     | DP_CCTALK_INTERFACE
                                     | DP_COIN_ACCEPT_DEVICE,

                                        // ICT Devices
    DP_ICT_U85                  =  1 | DP_MANU_ICT
                                     | DP_CCTALK_INTERFACE
                                     | DP_COIN_ACCEPT_DEVICE,

                                        // JCM Devices
    DP_JCM_CC_EBA               =  0 | DP_MANU_JCM
                                     | DP_CCTALK_INTERFACE          // ON cctalk interface
                                     | DP_NOTE_ACCEPT_DEVICE,

    DP_JCM_CC_WBA               =  1 | DP_MANU_JCM
                                     | DP_CCTALK_INTERFACE
                                     | DP_NOTE_ACCEPT_DEVICE,

    DP_JCM_NOTE                 =  0 | DP_MANU_JCM
                                     | DP_JCM_INTERFACE
                                     | DP_NOTE_ACCEPT_DEVICE,

                                        // GPT Devices
    DP_GPT_NOTE                 =  0 | DP_MANU_GPT
                                     | DP_GPT_INTERFACE
                                     | DP_NOTE_ACCEPT_DEVICE,
} ;
```

12.6 Coin (Note) Routing.

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

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

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

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

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

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

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

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

12.7 Meters

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

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

12.8 CounterIncrement

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

void CounterIncrement(long CounterNo,
long Increment);

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

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

Return Value
None

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

12.9 CounterCaption

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

void CounterCaption(long CounterNo,
char* Caption);

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

Return Value
None

Remarks
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.

If the counter specified is higher than the highest supported, then the call is silently ignored.
The specified caption is **not** stored in the meter, even if the meter offers this facility.

12.10 CounterRead

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

long CounterRead(long CounterNo);

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

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

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

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

12.11 ReadCounterCaption

Synopsis
The Read**CounterCaption** call is used to determine the caption for the specified counter
char* CounterCaption(long CounterNo);

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

Return Value
None

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

12.12 CounterDisplay

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

void CounterDisplay (long DisplayCode) ;

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

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

Return Value
None

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

12.13 MeterStatus

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

long MeterStatus (void);

Parameters
None

Return Value
One of the following:

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

Remarks
None

12.14 MeterSerialNo

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

long MeterSerialNo (void);

Parameters
None

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

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

12.15 E^{2}PROM

Included in the IMHEI card is E^{2}PROM memory, which is used by the embedded process to maintain counters etc. 256 bytes of this E^{2}PROM 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 E^{2}PROM memory on the card, after testing and before delivery to an end user.

12.16 E2PromReset

Synopsis
The **E2PromReset** call is made by the PC application software to clear all the E^{2}PROM memory on the card.

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

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

Return Value
None

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

12.17 E2PromWrite

Synopsis
The **E2PromWrite** call is made by the PC application software to write to all or part of the user E^{2}PROM on the card.

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

Parameters
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.
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
This call schedules the write to the E^{2}PROM memory and returns immediately. There is no way of knowing when the E^{2}PROM has actually been updated but, barring hardware errors, it will be complete within one second of the call.

12.18 E2PromRead

Synopsis
The **E2PromRead** call is made by the PC application software to obtain all or part of the user E^2PROM from the card.

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

Parameters
UserBuffer
This is the address of the user's buffer, into which the current contents of the user E^2PROM area are copied.
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
Unwritten E^2Prom memory is initialised all one bits.
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 E^2Prom memory.

13. API Software Guide - Engineering Support

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

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

13.1 WriteInterfaceBlock

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

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

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

Parameters
Interface

The serial number of the interface that is being accessed.

Block

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

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**.

13.2 ReadInterfaceBlock.

Synopsis
The **ReadInterfaceBlock** call reads the “raw” response to a single **WriteInterfaceBlock**.

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

Parameters
Interface
The serial number of the interface being accessed

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

Length
The space available in the program buffer.

Return Values

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

Remarks

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

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

14. Troubleshooting and support

14.1 Troubleshooting guide

Table 7: Troubleshooting guide

Error	Possible reason	Solution
No comms with card	Driver not running	Close all programs then run AESDriver.exe
	USB cable faulty	Change USB cable
	No +12V DC power	Turn on the power supply
Hoppers not shown in Dispenser screen	Wrong voltage hopper	Check that a 24V hopper is being used.
Lumina not shown in Acceptors screen	**PayLink** is not configured for the 6 digit security code	Run LuminaSerial.exe and change the 6 digit code
Ardac not shown in Acceptors screen	Wrong protocol in **PayLink**	Reprogram **PayLink** firmware to use desired protocol ID003 or Ardac 2

14.2 Support

For support using **PayLink**, please contact your local Money Controls Technical Services office.

Money Controls UK - Technical Services

Tel: +44 (0) 161 955 0124

E-mail: technical.uk@moneycontrols.com

Website: http://www.moneycontrols.com/support/technical_support.asp

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