****
****

Berg Generic Cash Register Interface Driver

Specification Version 2.00

0.10 Original Specification 03/12/90 MRS
0.20 Corrections in LRC examples, added code examples 12/19/90 MRS
1.0 Added emphasis -- escape sequence and pin 4 is optional 01/21/02 RFF
2.0 Used consistent terminology; corrected LRC example; add reference to pour with release; Change name from Standard to Generic; add handshake info; formatted and corrected code example; switch to automatic page numbering 07/09/03 DOW

The Berg Generic Cash Register Interface allows a Cash Register or POS system to interface with Berg dispensing units via a serial port for ringing drinks in an intelligent manner. By installing or enabling the Berg Generic driver in the Berg dispensing unit and complying with the following specification, the Berg dispensing system will send PLU (price look up) information to a cash register and optionally wait for acknowledgement from the register prior to dispensing.

A Berg dispensing unit will be referred to as an ECU (electronic control unit).

Different manufacturers may refer to their sales terminals as ECR, POS, Point of Sale or other terminology. This specification will use POS to refer to any sales terminal capable of interfacing to a Berg ECU dispenser unit via a serial port.

PLU is a numeric string that is used to indicate what was dispensed. It is known in some POS systems as NLU.

ELECTRONIC SPECIFICATION

The Berg ECU will communicate with a POS via a RS-232 data link with the following parameters:

Baud Rate : 2400 bps
Data Bits : 8
Parity : None
Stop Bits : 1
Handshaking : DTR

The only lines always used by the Berg ECU are **TxD**, **RxD**, and **GND**, with **DTR** sometimes used. These are defined as follows:

TxD : TRANSMIT DATA
Data transmitted from the Berg ECU

RxD : RECEIVE DATA
Data received by the Berg ECU

DTR : DATA TERMINAL READY
Output from ECU to signal that BERG interface driver is active.
If the BERG interface driver is active, this signal will be active.

NOTE: This is optional, the interface works fine without it.

GND : SIGNAL GROUND

The RS232 Connector of the Berg ECU is a 9 Pin D-type male connector with pin definitions as follows:

PIN	Signal		
1	**Data Carrier Detect**	**(DCD)**	**INPUT**
2	**Receive Data**	**(RxD)**	**INPUT**
3	**Transmit Data**	**(TxD)**	**OUTPUT**
4	**Data Terminal Ready**	**(DTR)**	**OUTPUT**
5	**Signal Ground**	**(GND)**	
6	**Data Set Ready**	**(DSR)**	**INPUT**
7	**Request To Send**	**(RTS)**	**OUTPUT**
8	**Clear To Send**	**(CTS)**	**INPUT**
9	**Ring Indicator**	**(RI)**	**INPUT**

Caution: It is important that only pins 2, 3, 4, and 5 be used by the register system interfacing to the Infinity CPU. Connecting other pins may cause incorrect operation of the ECR interface or damage the Infinity CPU. Also note that using pin 4 is optional. The interface works very well without it.

DATA FORMAT: ECU to POS

The Berg ECU will send a variable length packet of data prior to dispensing a drink. This packet contains Start (*STX*) and End (*ETX*) of transmission bytes as well as a PLU number in ASCII, modifiers and trailers, and a Logical Redundancy Check byte (*LRC*). The **STX**, **ETX**, **PLU**, and **LRC** are always sent in a packet but the modifier and trailer bytes are optional. The transmission packet will look as follows with items in ()'s being optional.

STX (Modifiers) PLU (Trailers) LRC ETX
NOTE: Failure to correctly implement the escape sequence detailed on the lower half of page three and in the EXAMPLE LRC CALCULATIONS on pages 4 and 5 will cause many PLUs to be inoperative.

FIELD DEFINITIONS:

STX 02H Start of Transmission

This byte signals the start of a packet of data.

PLU Price Look Up

The PLU field is a 1 byte or longer sequence which signifies what is being dispensed. The length of the PLU is the number of digits needed to express the PLU. There is no zero padding. A PLU cannot be zero. The range of PLU values varies depending on the type of Berg ECU. These values are sent out as an ASCII sequence, most significant byte first, according to the following table:

PLU – ASCII Mapping Table

Numerical Value	ASCII Value in Hex
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H

Example : PLU = 4598 Packet Data = 34H 35H 39H 38H

MODIFIER AND TRAILER FIELDS **NOTE: Failure to correctly implement the Escape Sequence for these fields will cause many PLUs to be inoperative.**

Modifiers and trailer bytes may take any value from 01H through FFH. The only byte value that is not allowed in these fields is the NULL value 00H. These fields are used to modify the action on a PLU or cause an action to occur. For example, a modifier with a value of 05H may be used to signal that the current drink is a small portion of a given PLU value. A trailer byte of 0DH might be used to indicate that the drink should be entered into the register. Modifiers and trailers are optionally programmed at the Berg ECU and it is up to the designer of the POS system to decide whether they may be useful to his or her system and the significance of the values. Most existing systems do not use the modifier and trailer information.

Some Berg ECUs can be configured to send different modifiers or trailers based on price level or size. The same modifier and trailer sequence will be sent for all pours dispensed as price level X and portion size Y. For a BERG ECU that supports three price levels (A, B, C) and three sizes (S, R, L), this leads to 9 distinct Modifier and Trailer sequences.

Other modifiers and trailers will be the same for all pours.

LRC Logical Redundancy Check

NOTE: Failure to correctly implement the Escape Sequence for this field will cause many PLUs to be inoperative.

This one byte value is calculated by performing an XOR function on all of the bytes in the data stream other than ETX and the LRC itself. This means the STX, PLU, and all modifier and trailer bytes are included in the LRC calculation.

ETX 03H End of Transmission

This byte signals the end of the packet data.

ESCAPE CHARACTER 7FH

Since the LRC could calculate to a 02H or 03H, and since the Modifier and Trailer fields allow any binary character other than the NULL (00H) character, it is possible that a 02H or 03H may appear in one or more of these fields. If this happens the cash register may incorrectly interpret these as STX or ETX control fields. To prevent this from occurring these character are preceded by an escape character and have their most significant bit set by the Berg ECU upon transmission. Whenever the code 7FH occurs in a packet the register must interpret this as an escape sequence and mask the most significant bit of the next character in the packet. If a 7FH is to be sent by Infinity then 2 consecutive 7FH bytes will be sent in the packet. By using 7FH as the escape character the code needed to handle escape sequences is kept at a minimum. Whenever the register sees a 7FH it discards that character from the packet and sets the most significant bit of the next character in the packet to 0.

Escape Table

Character		Two Byte Escape Sequence	
STX	02H	7FH	82H
ETX	03H	7FH	83H
ESCAPE	7FH	7FH	7FH

The LRC is maintained on the escape sequence by including both characters in the LRC calculation. **It is possible that one of the escaped characters may appear in the LRC field. If this occurs the LRC value will also be escaped.** That is, if the LRC for a given packet calculates to a value of 03H, Infinity will send this value as the two byte escape sequence 7FH 83H.

NOTE: Failure to correctly implement the Escape Sequence will cause many PLUs to be inoperative.

EXAMPLE LRC CALCULATION

Example Data Packet #1

The data to send in this case is a PLU of 135 along with a modifier equal to 16H and a trailer of 21H. The LRC is not calculated yet so we leave it as a ?? value.

Logical Data Packet

STX	MODIFIER	PLU	TRAILER	LRC	ETX
02H	16H	135	21H	??	03H

Since the PLU must be sent as an ASCII value we expand the PLU 135 into a 3-byte sequence.

Preliminary Data to Send

STX	MODIFIER	PLU DATA	TRAILER	LRC	ETX
02H	16H	31H 33H 35H	21H	??	03H

Now we proceed with the LRC calculation:

Data Type	Hex	Binary			
STX	02H	00000010b	00H	00000000b	Starting LRC
			02H	00000010b	
Modifier	16H	00010110b	02H	00000010b	Intermediate LRC
			16H	00010110b	
PLU 135m (1)	31H	00110001b	14H	00010100b	Intermediate LRC
			31H	00110001b	
(3)	33H	00110011b	25H	00100101b	Intermediate LRC
			33H	00110011b	
(5)	35H	00110101b	16H	00010110b	Intermediate LRC
			35H	00110101b	
Trailer	21H	00100001b	23H	00100011b	Intermediate LRC
			21H	00100001b	
			02H	00000010b	Final LRC

Since the LRC calculates to 02H, the escape sequence is used.

Final Data to Send

STX	MODIFIER	PLU DATA	TRAILER	LRC	ETX
02H	16H	31H 33H 35H	21H	7FH 82H	03H

Example Data Packet #2

The data to send in this case is a PLU of 29 along with a modifier equal to 03H and a trailer of 7FH. The LRC is not calculated yet so we leave it as a ?? value.

Logical Data Packet

STX	MODIFIER	PLU	TRAILER	LRC	ETX
02H	03H29	7FH	??	03H	

Since the PLU must be sent as an ASCII value we expand the PLU 29 into a 2 byte sequence. In this case the values 03H and 7FH are values which must have the escape sequence applied.
Our Physical packet now looks as follows:

Preliminary Data to Send

STX	MODIFIER	PLU DATA	TRAILER	LRC	ETX
02H	7FH 83H	32H 39H	7FH 7FH	??	03H

Now we proceed with the LRC calculation:

Data Type	Hex	Binary			
STX	02H	00000010b	00H	00000000b	Starting LRC

				02H	00000010b	
Modifier	(ESC)	7FH	01111111b	02H	00000010b	Intermediate LRC
				7FH	01111111b	
Modifier	(03H)	83H	10000011b	7DH	01111101b	Intermediate LRC
				83H	10000011b	
PLU 29	(2)	32H	00110010b	FEH	11111110b	Intermediate LRC
				32H	00110010b	
	(9)	39H	00111001b	CCH	11001100b	Intermediate LRC
				39H	00111001b	
Trailer	(ESC)	7FH	01111111b	F5H	11110101b	Intermediate LRC
				7FH	01111111b	
Trailer	(7FH)	7FH	0111111b	8AH	10001010b	Intermediate LRC
				7FH	01111111b	
				F5H	11110101b	Final LRC

The LRC has now been calculated so the final data packet is as follows.

Final Data to Send

STX	MODIFIER	PLU DATA	TRAILER	LRC	ETX
02H	7FH 83H	32H 39H	7FH 7FH	F5H	03H

DATA FORMAT: POS to ECU

With the Berg Generic driver installed, the Berg ECU will send the above packet of data and then wait for a response from the POS system. This response may be one of two things. Either an acknowledge byte (ACK) or a negative acknowledge (NAK). It is the responsibility of the POS system to verify that the correct data has been received from the Berg ECU by checking the LRC byte. If the data is not correct, the POS must respond with a NAK. If the data is correct then the POS must check to see if this request for a drink contains a valid PLU. If it does not, the POS must respond with a NAK. The POS may also want to make sure that the register is ready to ring a drink on its terminal. If everything is correct on the register end then the register should respond with an ACK.

The Berg ECU may be set to pour with or without release.

If pour without release is set, the Berg ECU will pour as soon as the PLU packet has been sent regardless or the response from the POS. The packet will not be resent even if a NAK is received.

If pour with release is set, then the Berg ECU will not dispense the drink until an ACK has been received. If the ECU does not receive a response within a user-defined time, the Berg ECU will act as if it had received a NAK. At this point the ECU may retry or the bartender may again request the pour of a drink. **During the time from the start of packet to the reception of an ACK the Berg ECU will not pour. It is important that the POS designer keep this time as short as possible.**

ECR Response Values

Response	Hex Values
ACK	06H
NAK	15H

Code Example

The following code shows a sample implementation in C for receiving PLUs from the Berg ECU with the Generic driver installed via a serial port. The correct baud rate, parity and data and stop bits must be set. The code assumes that 3 functions will be supplied by the developer. These functions are as follows:

int byte_available(void)
Checks the status of the serial port. If a byte is available to be read it returns a 1 otherwise it returns a 0.

byte get_byte(void)
Reads a 1 byte value received by the serial port.

void put_byte(byte sendbyte)
Sends a 1 byte value out the serial port.

```
#define         byte                unsigned char
#define         ACK                 0x06
#define         NAK                 0x15
#define         STX                 0x02
#define         ETX                 0x03
#define         ESCAPE_BYTE         0x7F
#define         BUFFER_LEN          40      /*length of input buffer*/
#define         NO_PLU              0
#define         ERR_ETX_NO_STX      -1
#define         ERR_BAD_LRC         -2
#define         ERR_OVERFLOW        -3
#define         ERR_BAD_DATA        -4

/*By making the following two variables static we don't have to wait in get_plu() */
/*for a full packet to come to us. Instead we can pick up the rest of the         */
/*packet on subsequent calls to get-plu(). This is important if other             */
/*things such as checking user input from a keyboard or other processing          */
/*needs to be done.                                                               */

static unsigned int input_buff_ptr=0;       /*offset for placing raw data       */
static byte input_buffer[BUFFER_LEN];       /*holds raw input packet from Infinity*/

/*-------------------------------------------------------------------------------*/
```

```
/*int get_plu(int time_to_wait)                                                    */
/*                                                                                */
/*Retrieves a PLU packet from a Berg ECU with the Basic driver                    */
/*installed and returns the corresponding PLU number sent. It is the             */
/*responsibility of the calling routine to check the validity of the PLU         */
/*for the POS system and respond to the ECU with either an ACK or a              */
/*NAK. This routine does not expect modifiers and trailers and will                    */
/*interpret them as part of the PLU number field as long as they are also ASCII  */
/* numeric.                                                                      */
/*Special handling for modifiers and trailers can be added.                      */
/*                                                                               */
/*Parameters:                                                                    */
/*int time_to_wait – The number of times to loop in get_plu when                 */
/*                no character is available                                      */
/*returns:                                                                       */
/*       A positive integer value corresponding to the PLU sent                  */
/*       or one of the following codes:                                          */
/*       NO_PLU: no PLU data currently available                                 */
/*       ERR_ETX_NO_STX :Received an ETX with no corresponding STX               */
/*       ERR_BAD_LRC       :Received a packet with a bad LRC value               */
/*       ERR_OVERFLOW    :Infinity data overflowed the receive buffer            */
/*       ERR_BAD_DATA    :Bad value found in PLU data (not 30H-39H)              */
/*-------------------------------------------------------------------------------------------*/

int get_plu(int time_to_wait)
{

/*Declare local data variables.                */

unsigned int i;
unsigned int plu=0;
byte newbyte;                       /*this is the newest byte from the communications port*/
byte lrcbyte;
byte datbuff[BUFFER_LEN];                   /*this buffer will hold data that has been
processed*/
                                    /*by removing ESC bytes and converting escaped bytes*/
                                    /*back to their real values*/
unsigned int datptr;                /*offset for data buffer*/

unsigned int temp_wait_time;

temp_wait_time=time_to_wait;        /*set up the wait time*/

while (temp_wait_time)
        {               /* until we are done getting a packet, have an error*/
        /* or run out of time we have to wait for a packet*/

        if (!byte_available())              /* if there is no byte available from ECU*/
```

```
        temp_wait_time--;                       /* just decrement the wait time variable*/
else
        {
        temp_wait_time=time_to_wait;        /* have a byte so reset the wait time*/

        newbyte=get_byte();             /* read the new byte*/

        switch(newbyte)
                {
        case ETX:
                if (input_buffer[0]!=STX)
                        {
                        /* if the first byte in buffer is not STX then error*/
                        /* send NAK then return error*/
                        put_byte(NAK);
                        return(ERR_ETX_NO_STX);
                        }
                else
                        {
                        /* ok our input buffer is full of data and we have both*/
                        /* STX and ETX characters so we now process the data*/
                        i=0;
                        datptr=0;
                        lrcbyte=0;      /* initialize the lrc byte */
                        do
                                {
                                if (input_buffer[i]!=ESCAPE_BYTE)
                                        {
                                        /* if this is not an escaped character*/
                                        lrcbyte^=input_buffer[i];/* XOR byte into lrc*/

                                        /* don't add the STX char in data buffer*/
                                        if (i!=0)
                                                datbuff[datptr ++]=input_buffer[i];

                                        i ++;/* update the pointer*/
                                        }
                                else
                                        {
                                        /* this is an escaped character*/
                                        /* First make sure this byte belongs to the data
                                        packet and not the LRC byte. If (input_buff_ptr-i) is
                                        not> 1 then this ESCAPE actually belongs to the
                                        LRC and not as part of the send data*/

                                        if ((input_buff_ptr-i)>1)
                                                {
                                                /* ok xor in the 2 bytes into the LRC byte*/
```

```
                        lrcbyte^=input_buffer[i ++];
                        lrcbyte^=input_buffer[i];
                        /* modify the ESC'd byte and put it in the
                        data buffer*/
                        input_buffer[i]&=ESCAPE_BYTE;
                        datbuff[datptr ++]=input_buffer[i];
                        }
                else
                        {       /* WHOA! We ran into an ESC for
                                the LRC byte so end the loop*/
                        break;
                        }
                }
        } while (i<input_buff_ptr);

/* i is now pointing to the LRC byte in the input buffer*/
/* if it is escaped we need to take care of that case*/

if (input_buffer[i] = = ESCAPE_BYTE)
        {
        /* escaped LRC so update pointer and modify LRC*/
        i ++;
        input_buffer[i]&=ESCAPE_BYTE;
        }

if(lrcbyte!=input_buffer[i])
        {
        /* oops!! lrc not what we expected – send NAK and*/
        /* return an error                          */
        put_byte(NAK);
        return(ERR_BAD_LRC);
        }
else
        {
        /* everythings valid so convert PLU string to a */
        /* PLU number                               */
        /* first put a 0 at the end of the PLU data      */
        datbuff[datptr]=0;
        datptr=0;  /* sent data pointer to start of PLU data*/
        plu=0;     /* set plu value to 0*/
        while(datbuff [datptr])
                {
                if (datbuff[datptr]<0x30 || datbuff[datptr]>0x39)
                        return(ERR_BAD_DATA);
                plu*=10;/* multiply by 10 to put in correct place*/
                plu+=datbuff[datptr]-0x30;/* subtract ascii offset*/
                datptr ++;              /* update data pointer*/
                }
```

```
                                    return(plu); /* return PLU back to the caller*/
                                    }
                            }
                    break;
            case STX:    /*STX…just set input buffer pointer to 0*/
                    input_buff_ptr=0;
                    break;
            default:        /* update buffer pointer to place new incoming character*/
                     input_buff_ptr ++;
                     if (input_buff_ptr = = BUFFER_LEN)
                            {
                            /* EEK BUFFER OVERFLOW!*/
                            input_buff_ptr = 0;
                            return(ERR_OVERFLOW);
                            }
                    break;
                    }  /* end switch */
            /* insert the new incoming character in the input buffer */
            input_buffer[input_buff_ptr] =newbyte;
            }  /* end byte available */
        } /* end while temp_wait_time */

return(NO_PLU);
}
```