1784-KTx Scanner
(Catalog Number 1784-KTX, -KTXD, and -KTS)

Use these release notes with publication 1784-6.5.20, the 1784-KT*x* Scanner Reference Manual.

Outlined boxes and change bars call your attention to each modification. We selected this format so you can cut and paste these blocks of information into your manual, if you wish.

On page P-1, add this new section about terminology to the preface.

Terminology That We Use

We use these abbreviations, acronyms, and initialisms in the scanner text.

Term	Meaning
ADS	adapter decide state (bit)
API	application programming interface
ASIC	application-specific integrated circuit
BT	block transfer
COS	change of state
EOI	end of interrupt
FDG	fault dependent group
ISR	interrupt service routine
MCB	module control byte
MDL	module decide length
PIC	programmable interrupt controller
SIF	scanner issued fault (bit)

On page P-1, note the addition of the README statement.

About the Examples

We designed the examples in this manual to introduce the application programmer to the host-to-KT*x* interface and commands. The examples demonstrate all of the host-to-KT*x* commands as well as other common tasks, such as:

- reading and writing the I/O image tables
- reading the status table
- start-up and shutdown
- watchdog timer

The examples of the host-to-KT*x* commands are designed to issue a command, then poll the KT*x* for a confirmation before issuing the next command. Some commands, such as block transfers, are given 4 seconds by the KT*x* to complete their tasks before the KT*x* declares a timeout error. For most applications it would be impractical to suspend processing for this length of time. Instead most applications should be designed to take advantage of the KT*x*'s ability to interrupt the host whenever a command confirmation is available or when other important events occur.

Source code for an interrupt service routine has been included to demonstrate how an application might handle KT*x* interrupts. The interrupt service routine has not been integrated with the rest of the example source code. That task has been left to the application programmer.

README: These examples are provided "AS IS" and for your convenience, without any express implied warranties of merchandise and fitness.

On page 2-11, note the addition of the protocol file name, SCANNER.BIN.

Loading the Protocol File

To download the protocol, your driver must:

- transfer the loader file to the dual port
- load the protocol file, SCANNER.BIN

Transferring the Loader File to the Dual Port

First transfer the loader binary file (LOADPCL.BIN) into the specified area of the dual-port RAM. Your driver uses the loader program to download the protocol ("soft" firmware) through the dual-port interface and into the Z80's RAM.

1. Clear all RAM on the KT*x* card.
   - **A.** Reset the KT*x* card by writing 01h to byte :803h.
   - **B.** Copy the CLRRAM.BIN program to the dual port, starting at :0000h.
   - **C.** Write **01h** to byte :0802h (to release the Z80).
   - **D.** Wait for location :002Ah to change to zero. If after two seconds this has not occurred, then the CLRRAM.BIN did not succeed.
   - **E.** Reset the KTx card by writing **01h** to byte :0803h.
2. Transfer the loader program (LOADPCL.BIN) into the dual port, starting at :0000h.

Next, download the protocol, SCANNER.BIN.

On page 4-11, note that the entry 'confirmation_length' in the Get Scan List Confirmation table has been modified.

Get Scan List

Use the Get Scan List command to obtain a copy of the current scan list from the scanner. This command is most useful after an autoconfiguration has been completed because it lets you see what adapters were found on the link.

Command Syntax

host_command	7
transaction_number	0 – 255
command_length	0

This command queues a confirmation immediately. The confirmation looks like this:

Get Scan List Confirmation

host_command	7
transaction_number	0 – 255 (whatever host supplied)
confirmation_status	SUCCESS ①
confirmation_length	# of entries in scan list + 1
scan_list_count	# of entries in scan list
adapter_address [64]	array of adapter addresses to be scanned 1 through 64 entries

① SUCCESS is the only possible condition.

The routine in Example 4.E instructs the scanner to return the current scan list.

On page 5-2, replace the section Adapter Decide Length (ADL) with the section shown below, Module Decide Length.

Module Decide Length

For most operations, the host dictates to the BT module how much data is to be transferred in a BT Read or Write. There are circumstances, such as when first configuring a BT module, when the host allows the BT module to specify the number of words to be transferred. This mode is called Module Decide Length.

To perform a Module Decide Length block transfer, the host issues a Host BT Read or Host BT Write to the scanner but specifies a bt_data_len of zero. If it is a Host BT Write, the host **must** supply 64 words of data and a data checksum to be transferred. If there are fewer than 64 words in bt_write_data in the command buffer, the scanner returns a confirmation with the error code:

HOST_SUPPLIED_TOO_LITTLE_DATA

On page 5-2, note the addition of a data checksum and its formula to the items the host must supply.

Host BT Write

The host application issues a Host BT Write command when it wants to initiate a block transfer to an adapter. To match the response to the request on completion of the block transfer, the host must supply:

- the *logical rack* address
- the block-transfer module slot address
- the data to write to the module
- a unique transaction number
- a data checksum (1's complement of the 8-bit addition of the data bytes in the block transfer)

As many as 64 block-transfer commands can be queued at one time, with the limitation of one BT per block-transfer module (as many as 16 BTs per logical rack). The scanner sets a 4-second timer for each BT request. If the timer expires before the BT exchange with the adapter completes, the BT request is cancelled and a confirmation is returned to the host indicating there was a BT timeout.

On page 5-3, note that the formula for command_length has been modified.

Command Syntax

host_command	4
transaction_number	0 – 255
command_length	4 + (2 * number of data words)
module_slot_address	0 – 15
logical_rack_address	0 – 31
bt_data_len	# of data words①
bt_write_data	1 – 64 words of data
bt_data_checksum	1's complement of the 8 bit addition of the data bytes in the block transfer

① You can specify 0 – 63; if you specify 0, the module decides the data length, and you must supply 64 words of data.

This command queues a confirmation when the block transfer exchange is completed, i.e., it has received a BT reply packet from the adapter, or when the 4-second timer has expired. The confirmation looks like:

Host BT Write Confirmation

host_command	4
transaction_number	0 – 255 (whatever host supplied)
confirmation_status	(see Table 5.A)
confirmation_length	0
data	N/A

On page 5-4, replace Table 5.A with the revised table below.

Table 5.A
Host BT Write Error Conditions

Error Mnemonic	Code	Description
SUCCESS	0	The BT Write was successful.
BAD_COMMAND_DATA_LENGTH	26	Command data length exceeds maximum.
BT_BAD_ADDRESS	27	Logical rack address or slot address is out of range.
BT_BAD_ADDRESS_NOT_IN_SCAN_LIST	28	BT address is not in scan list.
BT_BAD_DATA_LENGTH	29	More than 63 words were specified.
TOO_MANY_REQUESTS_FOR_MODULE	30	This module already has a BT in progress.
BT_QUEUE_FULL	31	Already 64 BT requests pending.
BT_REQUEST_TIMEOUT	36	BT exchange between scanner and adapter did not complete within 4 seconds.
USER_MODULE_REQUEST_TYPE_MISMATCH	38	Host requested a write, module responded with a read request.
USER_MODULE_LENGTH_MISMATCH	39	Length mismatch between what the scanner requested and the adapter requested.
HOST_SUPPLIED_TOO_LITTLE_DATA	40	Host did not supply 64 words of data for a module decide length BT.
BT_CHECKSUM_ERROR	41	Host-supplied checksum is invalid.
MODULE_IN_FAULT_GROUP_REQUESTED_BT	50	A BT Write was requested for a module that is in a faulted group.

The routine in Example 5.A instructs the scanner to perform a Host BT Write.

On page 5-5, replace Example 5.A with the revised example below.

Example 5.A

```
#include ”ktx_dp.h”       /* 1784–KTX Scanner Dualport definition            */
#include ”ktx_err.h”      /* 1784–KTX Scanner error codes                    */
#include ”ktxconst.h”     /* 1784–KTX Scanner constants                      */

UBYTE bt_write (KTX_DUALPORT far *dp, UBYTE slot_number, UBYTE link_address,
                          BT_BUFFER *bt_buf, UBYTE trans_num, USHORT timeout)
{
UBYTE        status, chk_sum;
int          i,size;

/**** Initialize the command buffer ****/

dp–>cmd_buffer.host_command = BT_WRITE;
dp–>cmd_buffer.transaction_num = trans_num;

/**** Copy entire BT buffer if length is ****/
/**** set to ADAPTER_DECIDE_LENGTH       ****/

if (bt_buf–>count == ADAPTER_DECIDE_LENGTH)
    size = 64;
else
    size = bt_buf–>count;
dp–>cmd_buffer.command_length = 4 + (2*size);
dp–>cmd_buffer.cmd.bt_write.module_slot_address = slot_number;

/**** Convert link address to logical rack address ****/

dp–>cmd_buffer.cmd.bt_write.logical_rack_address = link_address >> 2;
dp–>cmd_buffer.cmd.bt_write.bt_data_length = bt_buf–>count;
for (i=0, chk_sum=0; i<size; i++){
    dp–>cmd_buffer.cmd.bt_write.bt_data[i] = bt_buf–>bt_data[i];
    chk_sum += (UBYTE)(bt_buf–>bt_data[i]);
    chk_sum += (UBYTE)(bt_buf–>bt_data[i]>>8);
}
*((UBYTE *)&dp–>cmd_buffer.cmd.bt_write.bt_data[i]) = ~chk_sum;

/**** Send the command ****/

dp–>int_status_from_host = SCANNER_COMMAND_FROM_HOST;
dp–>host_to_ktx_int_reg = INTERRUPT_KTX;

/**** Get and acknowledge the confirmation  ****/

status = get_confirmation(dp, trans_num, timeout);

if ((status != CONFIRMATION_TIMED_OUT) && (status != TRANS_NUM_MISMATCH))
    acknowledge_confirmation(dp);

return status;
}
```

On page 5-6, note that the footnote in the Command Syntax table has been modified. Also note the modification of confirmation_length in the Host BT Read Confirmation table.

Host BT Read

The host application issues a Host BT Read command when it wants to initiate a block transfer from an adapter. To match the response to the request on completion of the block transfer, the host must supply:

- the *logical rack* address
- the block-transfer module slot address
- the data size to read from the module
- a unique transaction number

As many as 64 block-transfer commands can be queued at one time, with the limitation of one BT per block-transfer module (as many as 16 BTs per logical rack). The scanner sets a 4-second timer for each BT request. If the timer expires before the BT exchange with the adapter completes, the BT request is cancelled and a confirmation is returned to the host indicating there was a BT timeout.

Command Syntax

host_command	5
transaction_number	0 – 255
command_length	3
module_slot_address	0 – 15
logical_rack_address	0 – 31
bt_data_len	# of data words①

① You can specify 0 – 63; if you specify 0, the module decides the data length.

This command queues a confirmation when the block-transfer exchange is completed, i.e., it has received a BT reply packet from the adapter, or when the 4-second timer has expired. The confirmation looks like:

Host BT Read Confirmation

host_command	5
transaction_number	0 – 255 (whatever host supplied)
confirmation_status	(see Table 5.B)
confirmation_length	successful read: # of data bytes + 1 byte checksum unsuccessful read: 0 bytes
bt_read_data	1 – 64 words of data

Important: Version 6 of the KTX scanner binary (and all subsequent versions) will pass the block-transfer read checksum to the host in the host BT read confirmation.

On page 5-8, replace Example 5.B with the revised example below.

Example 5.B

```
#include ”ktx_dp.h”       /* 1784–KTX Scanner Dualport definition            */
#include ”ktx_err.h”      /* 1784–KTX Scanner error codes                    */
#include ”ktxconst.h”     /* 1784–KTX Scanner constants                      */

UBYTE bt_read (KTX_DUALPORT far *dp, UBYTE slot_number, UBYTE link_address,
                         BT_BUFFER *bt_buf, UBYTE trans_num, USHORT timeout)
{
int          i;
UBYTE        status, chk_sum;

/**** Initialize the command buffer ****/

dp–>cmd_buffer.host_command = BT_READ;
dp–>cmd_buffer.transaction_num = trans_num;
dp–>cmd_buffer.command_length = 3;
dp–>cmd_buffer.cmd.bt_read.module_slot_address = slot_number;

/**** Convert the link address to logical rack address ****/

dp–>cmd_buffer.cmd.bt_read.logical_rack_address = link_address >> 2;
dp–>cmd_buffer.cmd.bt_read.bt_data_length = bt_buf–>count;

/**** Send the command ****/

dp–>int_status_from_host = SCANNER_COMMAND_FROM_HOST;
dp–>host_to_ktx_int_reg = INTERRUPT_KTX;

/**** Get and process the confirmation  ****/

status = get_confirmation(dp, trans_num, timeout);

switch (status) {

    /* break if no confirmation was available */

    case CONFIRMATION_TIMED_OUT:
             break;

    /* break if the transaction numbers didn’t match */

    case TRANS_NUM_MISMATCH:
             break;

   /* when successful, copy BT read data to caller              */
   /* and acknowledge the confirmation                          */

    case SUCCESS:
     bt_buf–>count = dp–>confirmation_buffer.conf_length/2;
     for (i=0, chk_sum=0; i < bt_buf–>count; i++){
         bt_buf–>bt_data[i] =dp–>confirmation_buffer.conf.bt_read.bt_data[i];
         chk_sum += (UBYTE)(bt_buf–>bt_data[i]);
         chk_sum += (UBYTE)(bt_buf–>bt_data[i]>>8);
         }
     chk_sum = ~chk_sum;
     if (chk_sum != (UBYTE)(dp–>confirmation_buffer.conf.bt_read.bt_data[i])){
         /* ADD CALL / ERROR HANDLER FOR CHECK SUM ERROR */
         }
     acknowledge_confirmation(dp);
     break;

    /* if an error occurred, acknowledge the confirmation        */
    /* and return the status to the caller                       */

    default:
             acknowledge_confirmation(dp);
             break;

}
return status;
}
```

On page B-2, note the addition of the README statement.

About the Examples

We designed the examples in this manual to introduce the application programmer to the host-to-KT*x* interface and commands. The examples demonstrate all of the host-to-KT*x* commands as well as other common tasks, such as:

- reading and writing the I/O image tables
- reading the status table
- watchdog timer
- start-up and shutdown

The examples of the host-to-KT*x* commands are designed to issue a command, then poll the KT*x* for a confirmation before issuing the next command. Some commands, such as block transfers, are given 4 seconds by the KT*x* to complete their tasks before the KT*x* declares a timeout error. For most applications it would be impractical to suspend processing for this length of time. Instead most applications should be designed to take advantage of the KT*x*'s ability to interrupt the host whenever a command confirmation is available or when other important events occur.

Source code for an interrupt service routine has been included to demonstrate how an application might handle KT*x* interrupts. The interrupt service routine has not been integrated with the rest of the example source code. That task has been left to the application programmer.

README: These examples are provided "AS IS" and for your convenience, without any express implied warranties of merchandise and fitness.

On page B-3, replace the ktx_dp.h example file with the modified file below.

ktx_dp.h file

```
/************************************************************************
*  KTX_DP.H                                                             *
*                                                                       *
*  Description:                                                         *
*                                                                       *
*    This file contains definitions for all of the common structures     *
*    and type definitions used by the example 1784-KTX Scanner          *
*    software.                                                          *
*                                                                       *
*  History:                                                             *
*                                                                       *
*  02/22/93  MJG  Original creation.                                    *
*                                                                       *
************************************************************************/

#ifndef KTX_DP_H        /* Prevent multiple inclusions */
#define KTX_DP_H 1

/*=======================================================================
  IMPORTANT! Some compilers align structures and/or arrays to word boundaries
  unless instructed otherwise.  The following #pragma statement instructs the
  compiler to align to BYTE boundaries.
=======================================================================*/

#pragma pack(1)

/**** atomic data types ****/

#define UBYTE           unsigned char
#define BYTE            char

#define UWORD           unsigned short
#define WORD            short

#define USHORT          unsigned short
#define SHORT           short

#define UINT            unsigned int

#define ULONG           unsigned long
#define LONG            long

#define BOOL            int
#define METACHAR        short

/*=======================================================================

 CONFIG_DATA – defines the bits associated with the config byte.
                         The host can use this byte to control the behavior of
                         the scanner firmware.  The bits are active-high.

=======================================================================*/

typedef struct
{
             UBYTE        interrupt_after_each_scan:1;
             UBYTE        debug_flag:1;
             UBYTE        enable_cos_detect:1;
             UBYTE        undefined:5;
} CONFIG_DATA;

/*=======================================================================

 RUNNING_STATUS – defines the bits associated with the running status byte.
                        Used by the scanner firmware to notify the host of
running
                        status.
=======================================================================*/
```

```
typedef struct
{
             UBYTE        end_of_scan_list:1;
             UBYTE        adapter_addr_flt:1;
             UBYTE        undefined:6;
} RUNNING_STATUS;

/*===========================================================================

  OPERATING_STATUS   – this word is part of the dual port and contains Boolean
                       status information about the operation of
                       the scanner firmware.

===========================================================================*/

typedef struct
{
             UWORD               program_mode:1;
             UWORD               test_mode:1;
             UWORD               run_mode:1;
             UWORD               debug_flag:1;
             UWORD               unsolicited_bt:1;
             UWORD               bt_pending:1;
             UWORD               fault_exists:1;
             UWORD               fault_changed:1;
             UWORD               unsolicited_bt_read_reply:1;
             UWORD               unsolicited_bt_write_reply:1;
             UWORD               confirmation_queue_full:1;
             UWORD               unknown_interrupt_type:1;
             UWORD               host_dead:1;
             UWORD               cos_overrun:1;
             UWORD               undefined:2;
} OPERATING_STATUS;

/*===========================================================================

  ADAPTER_CONFIG_INFO   – this structure contains information about each
                          adapter on the Remote I/O link.  It is updated
                          during an autoconfiguration and when the host
                          changes the scan list.

===========================================================================*/

typedef struct
{
             UBYTE             in_scan:1;
             UBYTE             exists:1;
             UBYTE             known:1;
             UBYTE             size:2;
             UBYTE             node_adapter:1;
             UBYTE             undefined:2;
} ADAPTER_CONFIG_INFO;

/*===========================================================================

  ADAPTER_FAULT_INFO   – this structure contains fault configuration
                         and operating information and for an adapter.

===========================================================================*/

typedef struct
{
             UBYTE              fault_group_number:4;
             UBYTE              in_fault_group:1;
             UBYTE              adapter_online:1;
             UBYTE              adapter_group_faulted:1;
             UBYTE              undefined:1;
} ADAPTER_FAULT_INFO;

/*===========================================================================

  ADAPTER_STATUS   – this structure contains all the configuration
                                    and status information for an adapter.

===========================================================================*/

typedef struct
{
             ADAPTER_CONFIG_INFO           adapter_config_info;
             ADAPTER_FAULT_INFO            adapter_fault_info;
             UBYTE                         adapter_retry_count;
} ADAPTER_STATUS;
```

```
/*=============================================================================
                                    COMMAND DATA STRUCTURES
=============================================================================*/

typedef struct {
            UBYTE   scanner_mode;
} SET_MODE_CMD;

typedef struct {
            UBYTE   count;
            UBYTE   scan_list[64];
} SET_SCAN_LIST_CMD;

typedef SET_SCAN_LIST_CMD SCAN_LIST;

typedef struct {
            UBYTE   unused;
} AUTOCONFIGURE_CMD;

typedef struct {
            UBYTE   module_slot_address;
            UBYTE   logical_rack_address;
            UBYTE   bt_data_length;
            UWORD   bt_data[64];
            UBYTE   byte_allocation_w;  /* Because the checksum is dynamically
                                        placed after the last data word, this
                                        is only a placeholder.*/
} BT_WRITE_CMD;

typedef struct {
            UBYTE   module_slot_address;
            UBYTE   logical_rack_address;
            UBYTE   bt_data_length;
} BT_READ_CMD;

typedef struct {
            UBYTE   fg_number:4;
            UBYTE   in_fault_group:1;
            UBYTE   unused:3;
} FAULT_GROUP_BYTE;

typedef struct {
            FAULT_GROUP_BYTE        fault_group_data[128];
} SET_FAULT_GROUP_CMD;

typedef struct {
            UBYTE   unused;
} GET_SCAN_LIST_CMD;

typedef struct {
            UBYTE   host_command;
            UBYTE   transaction_num;
            UBYTE   command_length;
            union   {
                        SET_MODE_CMD            set_mode;
                        SET_SCAN_LIST_CMD       set_scan_list;
                        AUTOCONFIGURE_CMD       autoconfig;
                        BT_WRITE_CMD            bt_write;
                        BT_READ_CMD             bt_read;
                        SET_FAULT_GROUP_CMD     set_fault_group;
                        GET_SCAN_LIST_CMD       get_scan_list;
                        UBYTE                   padding[253];
                        } cmd;
} COMMAND;

/*=============================================================================
                                    CONFIRMATION DATA STRUCTURES
=============================================================================*/

typedef struct  {
            UBYTE   count;
            UBYTE   scan_list[64];
} GET_LIST_CONF;

typedef struct  {
            UBYTE   orphan;
} SET_LIST_CONF;
```

```
typedef struct  {
            UWORD   bt_data[64];
            UBYTE   byte_allocation_r;  /* Because the checksum is dynamically
                                        placed after the last data word, this
                                        is only a placeholder.*/
} BT_READ_CONF;

typedef struct
{
            UBYTE   host_command;
            UBYTE   transaction_num;
            UBYTE   conf_status;
            UBYTE   conf_length;
            union   {
                        GET_LIST_CONF   get_list;
                        SET_LIST_CONF   set_list;
                        BT_READ_CONF    bt_read;
                        UBYTE           dummy[252];
            } conf;
} CONFIRMATION;

/*============================================================================

  KTX_DUALPORT   – defines the structure of the dual port memory (shared by
                        both the PC Host and the KTX Z84 microprocessor).

============================================================================*/

typedef struct
{
             UBYTE              boot_code[4];
             UBYTE              link_state;
             UBYTE              link_address;
             UBYTE              link_protocol;
             UBYTE              link_baud_rate;
             UBYTE              reserved1;
             CONFIG_DATA        config_data;
             UBYTE              init_status;
             RUNNING_STATUS     running_status;
             UBYTE              int_status_to_host;
             UBYTE              int_status_from_host;
             UBYTE              host_dead_counter;
             UBYTE              reset_diags_counters;
             UBYTE              reserved2[2];
             UBYTE              alive_flag;
             UBYTE              duplicate_node;
             UBYTE              off_ktx;
             UBYTE              stopped_flag;
             UBYTE              module_state;
             UBYTE              COS_link_address;
             UBYTE              reserved3[100];
             UBYTE              num_adapter_addresses;
             UBYTE              num_faulted_adapters;
             OPERATING_STATUS   operating_status;
             ADAPTER_STATUS     adapter_status_table[128];
             COMMAND            cmd_buffer;
             CONFIRMATION       confirmation_buffer;
             UWORD              input_image_table[256];
             UWORD              output_image_table[256];

            /*
            **      The following are KTX hardware registers.  The numbers in
            **      the right column are hex offsets from the start of the
            **      dualport.
            */
```

```
            UBYTE             reserved_reg_1;            /* :800 */
            UBYTE             host_to_ktx_int_reg;       /* :801 */
            UBYTE             assert_reg;                /* :802 */
            UBYTE             deassert_reg;              /* :803 */
            UBYTE             status_reg;                /* :804 */
            UBYTE             reserved_reg_2;            /* :805 */
            UBYTE             reserved_reg_3;            /* :806 */
            UBYTE             reserved_reg_4;            /* :807 */
            UBYTE             host_ack_of_ktx_int_reg;   /* :808 */
            UBYTE             undefined_1;               /* :809 */
            UBYTE             card_control_write_reg;    /* :80A */
            UBYTE             undefined_2;               /* :80B */
            UBYTE             card_control_read_reg;     /* :80C */
            UBYTE             undefine_3;                /* :80D */
            UBYTE             reserved_reg_5;            /* :80E */
            UBYTE             undefined_4;               /* :80F */
} KTX_DUALPORT;

/*****************************************************************************
**
**                     NON-DUALPORT STRUCTURE DEFINITIONS
**
*****************************************************************************/

typedef struct  {
            UBYTE   count;
            UWORD   data[8];
} IO_BUFFER;

typedef UBYTE   TABLE_NAME;

typedef struct  {
            UBYTE   count;
            UWORD   bt_data[64];
} BT_BUFFER;

/*============================================================================
  IMPORTANT! Some compilers align structures and/or arrays to word boundaries
  unless instructed otherwise.  The following #pragma statement instructs the
  compiler to return to the alignment properties it was using before parsing
  this header file.
============================================================================*/

#pragma pack()

#endif
```

On page B-7, note that error message 40 now reads “Host_Supplied_Too_Little_Data.”

ktx_err.h file

```
/************************************************************************
*  KTX_ERR.H                                                            *
*                                                                       *
*  Description:                                                         *
*                                                                       *
*    This file contains all errors that are configured in               *
*    the 1784-KTX Scanner firmware.                                     *
*                                                                       *
*  History:                                                             *
*                                                                       *
*  07/08/93  MDE  Original creation.                                    *
*                                                                       *
************************************************************************/

#ifndef KTX_ERR_H
#define KTX_ERR_H 1

/*   init_status values   */

#define IN_PROGRESS                             1
#define INVALID_LINK_ADDRESS                    2
#define BINARY_PROTOCOL_MISMATCH                3
#define INVALID_BAUD_RATE                       4
#define UNAUTHORIZED_PROTOCOL                   5
#define FAILED_PROGRAM_CRC                      6

/*   general errors   */

#define SCANNER_NOT_PROGRAM                     15
#define SCAN_LIST_TOO_LONG                      16
#define SET_SCAN_LIST_IN_PROGRESS               17
#define AUTOCONFIGURING_IN_PROGRESS             18
#define SCAN_LIST_CAUSES_FDG_ORPHAN             19
#define SC_UNKNOWN_COMMAND                      20
#define SC_BAD_REQUEST                          21
#define SC_BAD_PARAM                            22
#define SC_CANNOT_CHANGE_MODE_DURING_SET_SCAN_LIST  23
#define SC_CANNOT_CHANGE_MODE_DURING_AUTOCONFIG     24
#define SCANNER_ALREADY_AUTOCONFIGURING         25
#define BAD_COMMAND_DATA_LENGTH                 26
#define BT_BAD_ADDRESS                          27
#define BT_BAD_ADDRESS_NOT_IN_SCAN_LIST         28
#define BT_BAD_DATA_LENGTH                      29
#define TOO_MANY_REQUESTS_FOR_MODULE            30
#define BT_QUEUE_FULL                           31
#define ADDRESS_NOT_IN_SCAN_LIST                32
#define CANNOT_SET_FG_WHILE_AUTOCONFIGURING     33
#define CANNOT_SET_FG_WHILE_SETUP_SCAN_LIST     34
#define BAD_CRC                                 35
#define BT_REQUEST_TIMEOUT                      36
#define UNSOLICITED_BT_REQUEST                  37
#define USER_MODULE_REQUEST_TYPE_MISMATCH       38
#define USER_MODULE_LENGTH_MISMATCH             39
#define HOST_SUPPLIED_TOO_LITTLE_DATA           40
#define BT_CHECKSUM_ERROR                       41
#define ADAPTER_ADDRESS_ERROR                   42
#define SCANNER_ADDRESS_ERROR                   43
#define ILLEGAL_REPLY_CMD                       44
#define BUFFER_OVERFLOW_ERROR                   45
#define TIMEOUT_ERROR                           46
#define BT_BLOWOFF                              47
#define IO_MASK_DATA_MISMATCH                   48
#define INVALID_ADAPTER_ADDRESS                 49
#define MODULE_IN_FAULT_GROUP_REQUESTED_BT      50
#define ADAPTER_SIZE_OVERLAP                    51
#define BAD_BTW_REPLY                           52
#define ERROR_MORE_THAN_32_UNIQUE_DEVICES       53
#define BAD_RAM                                 54
#define ADAPTER_CONFIG_SUCCESS                  129
#define TOO_MANY_ADAPTERS_ON_LINE               130

/**** Error codes for example routines ****/

#define      INITIALIZATION_TIMED_OUT           150
#define      CONFIRMATION_TIMED_OUT             151
#define      TRANS_NUM_MISMATCH                 152
#define      ADAPTER_NONEXISTENT                153
#define      UNKNOWN_IO_TABLE                   154

#endif
```

On page B-14, replace the confirm.c example file with the modified file below.

confirm.c file

```
/*****************************************************************************
**
**      UBYTE get_confirmation (KTX_DUALPORT far * dp,
**                      UBYTE trans_num, USHORT timeout)
**
**      This routine waits for <timeout> seconds for a confirmation to
**      become available.  It then validates the transaction number and
**      returns status to the calling routine.  It is the caller's
**      responsibility to acknowledge the confirmation.
**
**      INPUTS
**          KTX_DUALPORT far *dp – points to the base of the KTX dualport
**          UBYTE trans_num      – transaction_number
**          USHORT timeout       – time (in seconds) to wait for completion
**
**      OUTPUT
**          UBYTE status – completion status of the command
**
*****************************************************************************/

#include <time.h>
#include <stdlib.h>
#include <stdio.h>

#include "ktx_dp.h"     /* 1784–KTX Scanner Dualport definition */
#include "ktx_err.h"    /* 1784–KTX Scanner error codes         */
#include "ktxconst.h"   /* 1784–KTX Scanner constants           */

UBYTE   get_confirmation (KTX_DUALPORT far *dp, UBYTE trans_num, USHORT timeout)
{

/**** Initial values ****/

time_t  t0 = time(NULL);
UBYTE   status = SUCCESS;

/**** Wait for confirmation ****/

while (dp–>int_status_to_host != CONFIRMATION_INTERRUPT) {
    if ((USHORT)(time(NULL) – t0) > timeout) {
             status = CONFIRMATION_TIMED_OUT;
             break;
    }
}

/**** Check interrupt type ****/

switch (dp–>int_status_to_host) {

    /**** Handle non–confirmation types ****/

    case PROCESSING_PROBLEM_INTERRUPT :
             unrecoverable_error(dp);
             break;

    /**** Confirmations: match up transaction #'s ****/
    /**** If they match, return completion status ****/
    case CONFIRMATION_INTERRUPT :
             if (dp–>confirmation_buffer.transaction_num != trans_num)
                 status = TRANS_NUM_MISMATCH;
             else
                 status = dp–>confirmation_buffer.conf_status;

    default:
             break;
}

return status;
}
```

```
/*************************************************************************
**
**      void    acknowledge_confirmation (KTX_DUALPORT far * dp)
**
**      This routine performs the handshaking with the KTX to indicate
**      that the host has completed processing of the last confirmation.
**      It is the caller's responsibility to acknowledge the confirmation.
**
**      INPUTS
**          KTX_DUALPORT far *dp – points to the base of the KTX dualport
**
**      OUTPUT
**          none
**
*************************************************************************/

void    acknowledge_confirmation (KTX_DUALPORT far *dp)
{

/**** Initial values ****/

time_t  t0 = time(NULL);

/**** Wait for confirmation ****/

while (dp–>int_status_from_host != 0x00){
    if ((time(NULL) – t0) > 0x02){
             printf("\n KTX did not clear int_status_from_host in 2 seconds.\n");
             exit(FAIL);
    }
}

dp–>confirmation_buffer.host_command = 0;       /* zero confirmation buffer */
dp–>int_status_to_host = 0;                     /* zero interrupt type      */
dp–>host_ack_of_ktx_int_reg = ACKNOWLEDGE_KTX;  /* clears hardware int reg  */
dp–>int_status_from_host = CONFIRMATION_PROCESSED;  /* interrupt to KTX type */
dp–>host_to_ktx_int_reg = INTERRUPT_KTX;        /* interrupt the KTX        */

t0 = time(NULL);

while (dp–>int_status_from_host != 0x00){
    if ((time(NULL) – t0) > 0x02){
             printf("\n KTX did not clear int_status_from_host in 2 seconds.\n");
             exit(FAIL);
    }
}
}
```

On page B-23, replace the bt_write.c example file with the modified file below.

bt_write.c file

```
/*****************************************************************************
**
**        UBYTE bt_write (KTX_DUALPORT far * dp,
**                        UBYTE slot_number,
**                                    UBYTE link_address,
**                                    BT_BUFFER *bt_buf,
**                                    UBYTE trans_num,
**                                    USHORT timeout)
**
**        This routine instructs the KTX Scanner to perform a Block Transfer Write.
**
**        INPUTS
**            KTX_DUALPORT far *dp – points to the base of the KTX dualport
**          UBYTE slot_number          – BT module slot number (0–15)
**            UBYTE link_address       – device link address
**            BT_BUFFER *bt_buf        – BT data buffer
**            UBYTE trans_num          – transaction_number
**            USHORT timeout           – time (in seconds) to wait for completion
**
**        OUTPUT
**            UBYTE status             – completion status of the command
**
*****************************************************************************/

#include ”ktx_dp.h”       /* 1784–KTX Scanner Dualport definition            */
#include ”ktx_err.h”      /* 1784–KTX Scanner error codes                    */
#include ”ktxconst.h”     /* 1784–KTX Scanner constants                      */

UBYTE bt_write (KTX_DUALPORT far *dp, UBYTE slot_number, UBYTE link_address,
                        BT_BUFFER *bt_buf, UBYTE trans_num, USHORT timeout)
{
UBYTE        status, chk_sum;
int          i,size;

/**** Initialize the command buffer ****/

dp->cmd_buffer.host_command = BT_WRITE;
dp->cmd_buffer.transaction_num = trans_num;

/**** Copy entire BT buffer if length is ****/
/**** set to ADAPTER_DECIDE_LENGTH       ****/

if (bt_buf->count == ADAPTER_DECIDE_LENGTH)
    size = 64;
else
    size = bt_buf->count;
dp->cmd_buffer.command_length = 4 + (2*size);
dp->cmd_buffer.cmd.bt_write.module_slot_address = slot_number;

/**** Convert link address to logical rack address ****/

dp->cmd_buffer.cmd.bt_write.logical_rack_address = link_address >> 2;
dp->cmd_buffer.cmd.bt_write.bt_data_length = bt_buf->count;
for (i=0, chk_sum=0; i<size; i++){
    dp->cmd_buffer.cmd.bt_write.bt_data[i] = bt_buf->bt_data[i];
    chk_sum += (UBYTE)(bt_buf->bt_data[i]);
    chk_sum += (UBYTE)(bt_buf->bt_data[i]>>8);
}
*((UBYTE *)&dp->cmd_buffer.cmd.bt_write.bt_data[i]) = ~chk_sum;

/**** Send the command ****/

dp->int_status_from_host = SCANNER_COMMAND_FROM_HOST;
dp->host_to_ktx_int_reg = INTERRUPT_KTX;

/**** Get and acknowledge the confirmation  ****/

status = get_confirmation(dp, trans_num, timeout);

if ((status != CONFIRMATION_TIMED_OUT) && (status != TRANS_NUM_MISMATCH))
    acknowledge_confirmation(dp);

return status;
}
```

On page B-24, replace the bt_read.c example file with the modified file below.

bt_read.c file

```
/*****************************************************************************
**
**          UBYTE bt_read (KTX_DUALPORT far * dp,
**                      UBYTE slot_number,
**                                  UBYTE link_address,
**                                  BT_BUFFER *bt_buf,
**                                  UBYTE trans_num,
**                                  USHORT timeout)
**
**          This routine instructs the KTX Scanner to perform a Block Transfer
**          Read.
**
**          INPUTS
**              KTX_DUALPORT far *dp – points to the base of the KTX dualport
**             UBYTE slot_number    – BT module slot number (0–15)
**              UBYTE link_address   – device link address
**              BT_BUFFER *bt_buf  –  BT data buffer which is to be written on
**                                    completion of the BT
**              UBYTE trans_num –     transaction_number
**              USHORT timeout –      time (in seconds) to wait for completion
**
**          OUTPUT
**              UBYTE status – completion status of the command
**
*****************************************************************************/

#include ”ktx_dp.h”       /* 1784–KTX Scanner Dualport definition            */
#include ”ktx_err.h”      /* 1784–KTX Scanner error codes                    */
#include ”ktxconst.h”     /* 1784–KTX Scanner constants                      */

UBYTE bt_read (KTX_DUALPORT far *dp, UBYTE slot_number, UBYTE link_address,
                        BT_BUFFER *bt_buf, UBYTE trans_num, USHORT timeout)
{
int          i;
UBYTE        status, chk_sum;

/**** Initialize the command buffer ****/

dp–>cmd_buffer.host_command = BT_READ;
dp–>cmd_buffer.transaction_num = trans_num;
dp–>cmd_buffer.command_length = 3;
dp–>cmd_buffer.cmd.bt_read.module_slot_address = slot_number;

/**** Convert the link address to logical rack address ****/

dp–>cmd_buffer.cmd.bt_read.logical_rack_address = link_address >> 2;
dp–>cmd_buffer.cmd.bt_read.bt_data_length = bt_buf–>count;

/**** Send the command ****/

dp–>int_status_from_host = SCANNER_COMMAND_FROM_HOST;
dp–>host_to_ktx_int_reg = INTERRUPT_KTX;

/**** Get and process the confirmation  ****/

status = get_confirmation(dp, trans_num, timeout);

switch (status) {

    /* break if no confirmation was available */

    case CONFIRMATION_TIMED_OUT:
             break;

    /* break if the transaction numbers didn’t match */

    case TRANS_NUM_MISMATCH:
             break;

   /* when successful, copy BT read data to caller              */
   /* and acknowledge the confirmation                          */

    case SUCCESS:
     bt_buf–>count = dp–>confirmation_buffer.conf_length/2;
```

```
    for (i=0, chk_sum=0; i < bt_buf->count; i++){
        bt_buf->bt_data[i] =dp->confirmation_buffer.conf.bt_read.bt_data[i];
        chk_sum += (UBYTE)(bt_buf->bt_data[i]);
        chk_sum += (UBYTE)(bt_buf->bt_data[i]>>8);
        }
    chk_sum = ~chk_sum;
    if (chk_sum != (UBYTE)(dp->confirmation_buffer.conf.bt_read.bt_data[i])){
        /* ADD CALL / ERROR HANDLER FOR CHECK SUM ERROR */
        }
    acknowledge_confirmation(dp);
    break;

    /* if an error occurred, acknowledge the confirmation       */
    /* and return the status to the caller                      */

    default:
             acknowledge_confirmation(dp);
             break;

}
return status;
}
```

On page A-2, note the modification to the dual-port layout.

:00E	host_dead_counter	1			If zero, the host dead counter is disabled. A non–zero value defines the timeout period: 20ms * value The host must set the alive_flag = 0 before the timeout expires or the scanner will declare the host dead.
:00F	reserved2	3			
:012	alive_flag	1			See host_dead_counter.
:013	duplicate_node	1	R	W	Set to 1 if a duplicate scanner is detected.
:014	off_ktx	1			
:015	stopped_flag	1			
:016	module_state	1	R	W	If there is a runtime error (CRC or RAM test), the error code will be written here.
:017	cos_link_address	1	R/W	R/W	If the host enables CHANGE-OF-STATE detection, the link address of the adapter that had a change of state will be written here by the scanner. The host must clear this by writing an 0FFh.
:018	unused	64H			
:07C	num_adapter_addresses	1	R	R/W	Number of adapters on link
:07D	num_faulted_adapters	1	R	R/W	Number of adapters on link that are currently faulted
:07E	operating_status	2	R	R/W	This byte is used by the KT*x* to indicate to the host the state of the module by setting/clearing the bits. The bits (when set) are defined as: Bit 0: Program mode Bit 1: Test mode Bit 2: Run mode Bit 3: Debug mode Bit 4: Unsolicited BT received on link Bit 5: BT pending Bit 6: Fault exists Bit 7: Fault changed Bit 8: Unsolicited BT read reply Bit 9: Unsolicited BT write reply Bit 10: Confirmation queue is full Bit 11: Unknown host interrupt type Bit 12: Scanner declared host dead Bit 13: Input Image Table changed state Bits 14–15: unused

Publication 1784-6.5.20-RN2 – April 1996
Supersedes Publication 1784-6.5.20-RN1 – June 1995
PN 955124-69