HEWLETT
PACKARD

HP-CIO I/O EIGHT CHANNEL TERMINAL MULTIPLEXER

FIRMWARE INTERNAL MAINTENANCE SPECIFICATION (IMS)

PROJECT YUBA

27130A

HEWLETT-PACKARD COMPANY
Roseville Networks Division
8000 Foothills Boulevard
Roseville, California 95678

July 14, 1983

Terry Gong
Greg Dolkas

HEWLETT-PACKARD PRIVATE

HEWLETT
PACKARD

HP-CIO I/O EIGHT CHANNEL TERMINAL MULTIPLEXER

FIRMWARE INTERNAL MAINTENANCE SPECIFICATION (IMS)

PROJECT YUBA

HEWLETT-PACKARD COMPANY
Roseville Networks Division
8000 Foothills Boulevard
Roseville, California 95678

July 14, 1983

Terry Gong
Greg Dolkas

HEWLETT-PACKARD PRIVATE

History

date description

821111 Original document

830510 Complete document

830714 Added debug monitor to the product firmware for trouble shooting.

INTRODUCTION	CHAPTER 1

1.1 Scope

This document describes the internal structure of the firmware that as implemented for the HP-CIO RS-232-C/RS-422/3 8 channel terminal multiplexer card. Specially, this document describes:

- o all symbols used by the firmware except for symbols used as labels in the instruction sequence,
- o the firmware data structure,
- o the function of each firmware modules, and
- o any other information pertinent to the understanding of the firmware.

The reader is referred to the following related documents.

- o Zilog Z80 CPU Technical Manual.
- o Zilog Z80-SIO Technical Manual.
- o Zilog Z80-CTC Technical Manual.
- o CSG I/O EIGHT CHANNEL TERMINAL MULTIPLEXER, FIRMWARE EXTERNAL REFERENCE SPECIFICATION (ERS) by Greg Dolkas (RVD).
- o BACKPLANE INTERFACE CIRCUIT (BIC) ERS by Bill Martin, Roseville Division (RVD).
- o CSG STANDARD I/O BACKPLANE by R. B. Haagens, Computer Systems Division (CSY).
- o STANDARD DATACOM DRIVER by Greg Dolkas, Roseville Division (RVD).
- o CSG I/O STANDARD BACKPLANE PROTOCOL FOR SMART DATACOM

CARDS by Greg Dolkas, Roseville Division (RVD).

- o GUIDEBOOK TO DATA COMMUNICATIONS, Part No. 5955-1715, Hewlett-Packard, 1977.
- o HP-1000 SERIES, Z80 REAL TIME SCHEDULER, TECHNICAL SPECIFICATION by Tom Szolyga, June, 1979.

This document assumes the reader has the full understanding of all the information given in the MUX ERS.

Note that this document does not contain complete documentation of the self-test. Much of the code was leverage from the HP-CIO ASI and PSI self-test and from the MEF/L/A PSI self-test. The listing of the self-test subprogram contains most of the required documentation.

EQUATE & VARIABLE SYMBOLS DICTIONARY	CHAPTER 2

This chapter defines all the symbols which are not used as a label or subprogram name. However, all the symbols used in the self-test is not defined here because it is a self-contained module (that is, leverage from the ASI and PSI self-test). See the listing for documentation information.

Each global and local equate and variable symbols are defined in this chapter. Each symbol will have 3 or 4 attribute descriptions follow by the usage description.

The attribute descriptions are as follow:

- o The first attribute is whether the symbol is globally defined. The symbol is globally defined if it is used in an COPY file or if it is given the PUBLIC attribute. Otherwise, the symbol is locally defined within the module.
- o The second attribute is the type of the symbol which includes:
  - code if the symbol is defined in the CSEG area by using either DEFW or DEFB,
  - define label if the symbol is defined by using the DEFL pseudo op,
  - equate if the symbol is defined by using the EQU pseudo op,
  - FBIT if the symbol is defined by using the FBIT macro,
  - variable if the symbol is defined by using the DEFS pseudo op,
  - macro if the symbol is used to define offset to access the content .
- o The third attribute or information parameter is the name of the source file where the symbol is defined.
- o The fourth attribute is included only if the second attribute is defined as being a variable. This defined the

size of the variable.

ABD - global, equate, [MUX; BIC register 2 status code, abort done

ABORT - global, equate, [MUX; WTC order request code to abort a given transaction

ABRT_TBL - local, code, &BCWTC, 12 bytes; jump table to abort processing routines

ACTPRIO - global, variable, &MUXVR, 1 byte; contains the priority of the active transaction

ACTREQ - global, variable, &MUXVR, 2 bytes; contains the address to the active request block

ACTSTAT - global, variable, &MUXVR, 1 byte; contains the RTS code for the active transaction

ACTTID - global, variable, &MUXVR, 2 bytes; contains the transaction ID (TID) of the active transaction

AES - global, equate, [MUX; BIC registr 2 status code, asynchronous event

ALERT1 - global, equate, [MUX; bit assignment for alert 1 option in ALRT_OPT

ALOC - global, macro, [MUX; macro to assign offset value to a given symbol

ALOC_CNT - global, define label, [MUX; offset assignment to be used with the macro ALOC

ALRT_OPT - global, ALOC, [MUX, 1 byte; alert 1 option

ARQ_BUF - local, variable, &BPISR, 16 bytes, the FIFO buffer to queue ARQ status code for the host

ARQ_PTRA - local, variable, &BPISR, 2 bytes, the buffer pointer to ARQ_BUF for the writing the next ARQ status code

ARQ_PTRB - local, variable, &BPISR, 2 bytes, the buffer pointer to ARQ_BUF for the reading the next ARQ status code

BACK_SP - global, ALOC, [MUX, 1 byte; contains the character fo backspacing when edit mode is enabled

BAUD_RAT - global, ALOC, [MUX, 1 byte; contains the baud rate value

BAUD_TAB - local, code, &SPDSN, 17 bytes; the baud rate table to match the first zero's count which is the start bit, used in speed sensing

BEGSTACK - global, equate, [MUX; beginning address of the stack

BICEND - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the BIC END bit

BIC_0 - local, equate, &BPISR; I/O port address for BIC register 0

BIC_1 - local, equate, &BPISR; I/O port address for BIC register 1

BIC_2 - local, equate, &BPISR; I/O port address for BIC register 2

BIC_3 - local, equate, &BPISR; I/O port address for BIC register 3

BIC_4 - local, equate, &BPISR and &MUXMN; I/O port address for BIC register 4

BIC_5 - local, equate, &BPISR and &MUXMN; I/O port address for BIC register 5

BIC_6 - local, equate, &BPISR and &MUXMN; I/O port address for BIC register 6

BIC_ENI - local, equate, &BPISR; bit assignment for MIC register 0, set to enable the BIC interrupt

BLK_BIT - global, equate, [MUX; bit assignment for block mode in the WIC order request code

BLK_MASK - global, equate, [MUX; mask to get the block mode bit from the WIC order request code

BP_CMD - local, variable, &BPISR, 1 byte; contains the current command from the BIC register 1

BP_INT - local, variable, &BPISR, 1 byte; contains the current interrupt status from BIC register 5

BP_ORDER - global, variable, &BPISR, 1 byte; contains the current BIC order being processed

BPTX_PTR - global, ALOC, [MUX, 2 bytes; backplane transmit buffer pointer

BRK_ON - global, equate, [MUX; bit assignment for PORTSTAT+1 set when break detection is actively in progress

BRK_RX - global, equate, [MUX; bit assignment for INT_STAT set when

break event occurred

BS - global, equate, [MUX; ASCII backspace character

BSLASH - global, equate, [MUX; ASCII back slash character

BP_PTR - global, ALOC, [MUX, 1 byte; RX buffer location of the last backspace action

BUF_ADDR - local, variable, &BPISR, 2 bytes; the buffer address for continuing the data transfer when the buffer wrap around in the circular buffer

BUF_ADRW - local, variable, &BPISR, 2 bytes; contains the beginning address of the host write request for the bad BIC fix

BUF_LEN - local, variable, &BPISR, 2 bytes; the length of the data transfer for the wrap around buffer of the circular buffer data transfer

BUF_LENW - local, variable, &BPISR, 2 bytes; contains the data transfer length for the host write request for the bad BIC fix

BYTE - local, equate, &BPISR; bit assignment for BIC register 2, set when want byte wide data transfer

CAR_WREG - global, variable, &MUXVR, 1 byte; contains pseudo card write register content which defines the control for the card LED, self-test mode, hood LED, single-ended driver, and differential driver

CDC - local, equate, &BPISR; bit assignment for BIC register 3, the clear device clear bit

CHAR_LEN - global, ALOC, [MUX, 1 byte; contains the character length for the SIO

CH_AL1 - global, equate, [MUX; bit assignment for RX_FLAGS+1 set when alert 1 mode is enabled and no alert 1 has been sent yet

CH_ECHO - global, equate, [MUX; bit assignment for RX_FLAGS set when echoing is enabled

CH_EDIT - global, equate, [MUX; bit assignment for RX_FLAGS set when edit mode is enabled

CH_HAND - global, equate, [MUX; bit assignment for RX_FLAGS set when handshaking is enabled

CH_QUOT - global, equate, [MUX; bit assignment for RX_FLAGS set when quoting mode is enabled

CH_SIGN - global, equate, [MUX; bit assignment for RX_FLAGS set when signal character detection is enabled

CH_STT - global, equate, [MUX; bit assignment for RX_FLAGS set when single text termination is enabled

CMD - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the command bit

CMD_TBL - local, code, &BPISR, 32 bytes; jump table containing processing routine address which correspond to the command to be processed

CN_CARD - global, equate, [MUX; WIC order request code for control card request

CN_DEV - global, equate, [MUX; WIC order request code for control device request

CONF_BUF - global, ALOC, [MUX, CONF_SIZ bytes; the write card configuration staging buffer

CONF_SIZ - global, equate, [MUX; size of the write card configuration table which is currently 60 bytes

COUNT - global, equate, [MUX; bit assignment for RD_OPT set when the end-on-count option is enabled

COUNT - local, variable, &DMAA, 2 bytes; counter in the MIC channel A ISR to measure how many time the MIC failed by generating an interrupt for channel A DMA, the MUX card never uses the MIC channel A DMA

CR - global, equate, [MUX; ASCII carriage-return character

CTC0_TAB - local, code, &MXWCC, 34 bytes; CTC baud rate generator programming values, each entry consists of 2 bytes, the first byte is the control byte for the CTC, the second byte is the time constant

CTCIVEC0 - global, code, &MUXIV, 16 bytes; contains the interrupt service routine addresses for CTC #0

CTCVEC - global, equate, [MUX; low byte of the starting address for the CTC interrupt table

CTC_0_C0 - global, equate, [MUX; CTC #0 channel 0 I/O port address, the DMA pacer for the MIC DMA card or the firmware real

time clock for the Z80 DMA card

CTC_0_C1 - global, equate, [MUX; CTC #0 channel 1 I/O port address, port #1 baud rate generator

CTC_0_C2 - global, equate, [MUX; CTC #0 channel 2 I/O port address, port #0 baud rate generator

CTC_0_C3 - global, equate, [MUX; CTC #0 channel 3 I/O port address, the firmware real time clock for the MIC DMA card or the BIC interrupt generator for the Z80 DMA card

CTC_1_C0 - global, equate, [MUX; CTC #1 channel 0 I/O port address, port #2 baud rate generator

CTC_1_C1 - global, equate, [MUX; CTC #1 channel 1 I/O port address, port #3 baud rate generator

CTC_1_C2 - global, equate, [MUX; CTC #1 channel 2 I/O port address, port #4 baud rate generator

CTC_1_C3 - global, equate, [MUX; CTC #1 channel 3 I/O port address, unused

CTC_2_C0 - global, equate, [MUX; CTC #2 channel 0 I/O port address, port #5 baud rate generator

CTC_2_C1 - global, equate, [MUX; CTC #2 channel 1 I/O port address, port #6 baud rate generator

CTC_2_C2 - global, equate, [MUX; CTC #2 channel 2 I/O port address, port #7 baud rate generator

CTC_2_C3 - global, equate, [MUX; CTC #2 channel 3 I/O port address, unused

CTC_BAUD - global, ALOC, [MUX, 1 byte; contains the CTC I/O port address for the baud rate generator

CTC_TBL - local, code, &MUXMN, 8 bytes; table of CTC I/O port address for the baud rate generator corresponding to the port, index by the port number

C_OUTSP - global, equate, [MUX; bit assignment for UNIX_OPT set when the conditional output separator appendage option is enabled

DATALEN - global, ALOC, [MUX, 2 bytes; the data transfer length from the transaction request block of the WIC order

DATA_AVA - global, equate, [MUX; bit assignment for INT_STAT, set

when data is available in the receive buffer, i.e., a receive record is avaiable for the host

DATA_MSK - global, ALOC, [MUX, 2 bytes; the data and parity mask

DCL - local, eqaute, &BPISR; bit assignment for BIC register 3, the device clear bit

DEL - global, equate, [MUX; ASCII DEL character

DEVEND - local, equate, &BPISR; bit assignment for BIC register 4, the device end bit

DEV_HAND - global, ALOC, [MUX, 1 byte; the handshake options

DIFFDVR - global, equate, [MUX; bit assignment for CAR_WREG, set to disabled the front end differential driver

DIFFX - local, variable, &SPDSN, 1 byte; the tolerance band width to determine if the speed sensed value is within the potential baud rate

DLF - global, equate, [MUX; BIC register 2 status code, download failed

DMAB_SW - local, equate, &BPISR; bit assignment for MIC register 0, the DMA B switch bit, clear for backplane DMA, set for frontplane DMA

DMACT - local, equate, &MUXMN; the size of DMAI

DMAHRD - global, FBIT, [MUX; set when doing host read DMA with the Z80 DMA card

DMAHRDA - local, variable, &BPISR, 14 bytes; contains the Z80 DMA programming instructions for doing a host read

DMAHRDAR - local, variable, &BPISR, 2 bytes; this is part of array DMAHRDA, contains the buffer address for the data transfer from the card to the host

DMAHRDBL - local, variable, &BPISR, 2 bytes; this is part of array DMAHRDA, contains the buffer length for the dta transfer from the card to the host

DMAHRDCT - local, equate, &BPISR; the length of array DMAHRDA

DMAHWD - local, variable, &BPISR, 12 bytes; contains the Z80 DMA programming instructions for data transfer from the host to the card

DMAHWDAR - local, variable, &BPISR, 2 bytes; this is part of array DMAHWD, contains the buffer address for the data transfer from the host to the card

DMAHWDBL - local, variable, &BPISR, 2 bytes; this is part of array DMAHWD, contains the buffer length for the data transfer from the host to the card

DMAHWDCT - local, equate, &BPISR; the length of array DMAHWD

DMAI - local, code, &MUXMN, 7 bytes; programming instructions to initialize the Z80 DMA

DMAINIT - local, code, &BPISR, 26 bytes; programming instructions to initialize the Z80 DMA

DMAWRAP - global, FBIT, [MUX; set when have wrap around on the circular buffer when using the Z80 DMA card

DMA_DIR - local, equate, &BPISR; bit assignment for MIC registers 3 and 8, data transfer direction bit, clear for from memory, set for to memory

DMA_END - local, equate, &BPISR; bit assignment for MIC register 0, clear to assert END on the last byte in the host read data transfer

DMA_ENI - local, equate, &BPISR; bit assignment for MIC registers 3 and 8, set when the MIC is to generate an interrupt after the data transfer completes

DMA_ENO - local, equate, &BPISR; bit assignment for MIC registers 3 and 8, set to enable the DMA operation

DMA_MEM - local, equate, &BPISR; bit assignment for MIC registers 3 and 8, clear to increment for the memory address, set to decrement for the memory address

DOD - global, equate, [MUX; BIC register 2 status code to host, dead or dying

EAK - global, equate, [MUX; WTC order request code, event acknowledge

EBLEN - global, equate, [MUX; length of the echo buffer

ECHO - global, equate, [MUX; bit assignment for RD_OPT, set when for echoing the receive character

ECHOBUF - global, variable, &MUXVR, 1024 bytes; the starting address for the echo buffers for all 8 ports

ECHOPTRI - global, ALOC, [MUX, 2 bytes; contains the address for the next echo character to be inserted into the echo buffer

ECHOPTRO - global, ALOC, [MUX, 2 bytes; contains the address for the next echo character to be transmitted from the echo buffer

ECHO_SIN - global, equate, [MUX; bit assignment for DEV_HAND, set when the single text terminator is to be echoed

ECHO_TBL - local, code, &MUXMN, 16 bytes; table of echo buffer address for each port, index by the port number

ECH_CRLF - global, equate, [MUX; bit assignment for DEV_HAND, set when the single text terminator defined in SIN_TEXT is to cause the echoing of the CR-LF characters in place of it

EDIT - global, equate, [MUX; bit assignment for RD_OPT, set when the edit mode option is enabled

EDITCHAR - global, equate, [MUX; bit assignment for the special character tables P0SCHTBL, P1SCHTBL, ..., P7SCHTBL, set when the corresponding character is an edit character for editing

EHCTR - local, equate, &MUXMN; the host ENQ/ACK pacing counter default value

END - global, equate, [MUX; WTC order request code for the end-of-data

END_CT - global, ALOC, [MUX, 2 bytes; contains the end-on-count value from the host for the end-on-count option

ENQ_TIMR - global, ALOC, [MUX, 1 byte; the host ENQ/ACK or handshake timer value

EONCT - local, equate, &MUXMN; the default end-on-count value

EQ_DCNTR - global, ALOC, [MUX, 1 byte; the host ENQ/ACK character down counter

EVBITMSK - local, code, &MXWCC, 9 bytes; the bit mask corresponding the bit position in INT_STAT for the corresponding event

EVB_LEN - global, equate, [MUX; length of the event block

EVENTQ - global, variable, &MUXVR, 2 bytes; contains the link list adress of the pending event queue

EVNTABLE - local, code, &BCRTS, 32 bytes; a table of masks to remove the appropriate event which was sent to the host, each entry consists of 2 bytes

EVNT_BRK - global, equate, [MUX; event code for break received

EVNT_MSG - global, equate, [MUX; event code for data message available

EVNT_SC1 - global, equate, [MUX; event code for signal character 1 detected

EVNT_SC2 - global, equate, [MUX; event code for signal character 2 detected

EVNT_SC3 - global, equate, [MUX; event code for signal character 3 detected

EVNT_SC4 - global, equate, [MUX; event code for signal character 4 detected

EVNT_SSM - global, equate, [MUX; event code for speed sense mode completed

EVNT_TIM - global, equate, [MUX; event code for handshake timed out

EVNT_TX - global, equate, [MUX; event code for transmit buffer is empty

EVNUM - global, equate, [MUX; the maximum number of pending events per card

EVPRITBL - local, code, &MXWCC, 9 bytes; the event priority value corresponding to each event

EVQFREE - global, variable, &MUXVR, 2 bytes; contains the link list address to the free event blocks

EV_BLKS - global, variable, &MUXVR, EVNUM*EVB_LEN bytes (640 bytes); the event blocks storage area

EV_CODE - global, ALOC, [MUX, 1 byte; index into the event block for the event code if an event was sensed

EV_DISB - global, equate, [MUX; bit assignment for PORTSTAT+1, set when an event is sent to the host, it is cleared when an event acknowledge is received from the host with no more event on the queue

EV_LEN - global, ALOC, [MUX, 2 bytes; index into the event block containing the length of the message received buffer

EV_NEXT - global, ALOC, [MUX, 2 bytes; link address to the next event block

EV_PID - global, ALOC, [MUX, 1 byte; index into the event block containing the port ID number

EV_PRIO - global, ALOC, [MUX, 1 byte; index into the event block containing the event block priority

EV_QUED - global, equate, [MUX; bit assignment for PORTSTAT+1, set when an event is queued on the RTS queue for this port

EV_RTS_S - global, ALOC, [MUX, 1 byte; index into the event block containing the RTS status code

EV_TERM - global, ALOC, [MUX, 1 byte; index into the event block containing the text terminator character for the message received event

EV_TID - global, ALOC, [MUX, 2 bytes; index into the event block containing the transaction ID number, if any

EV_TYPE - global, ALOC, [MUX, 1 byte; index into the event block containing the terminating code for the message received event

FBIT - global, macro, [MUX; macro used to define the flag to be used with macros FCLR, FSET, and FTST

FCL - local, equate, &BPISR; bit assignment for BIC register 3, the fifo clear bit

FCLR - global, macro, [MUX; macro to clear the flag defined by using macro FBIT

FF - global, equate, [MUX; the character 0FFH

FFR - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the FIFO ready bit

FIFORBPI - local, equate, &BPISR; control word for BIC register 4 for read byte, pre-end, srq-immediate

FIFORWPI - local, equate, &BPISR; control word for BIC register 4 for read word, pre-end, srq-immediate

FIFOWBI - local, equate, &BPISR; control word for BIC register 4 for write byte, srq-immediate

FIFOWWI - local, equate, &BPISR; control word for BIC register 4 for write word, srq-immediate

FIFO_SRQ - local, equate, &BPISR; control word for BIC register 4 for SRQ for the next order

FILLER1 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER10 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER11 - global, ALOC, [MUX, 1 byte; filler in the port configuraion information

FILLER12 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER13 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER14 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER15 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER16 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER2 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER3 - global, ALOC, [MUX, 2 bytes; filler in the port configuration information

FILLER4 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER5 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER6 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FILLER7 - global, ALOC, [MUX, 1 byte; filler in the port configuration information

FLAGS - local, variable, &MUXVR, (FLBIT+7)/8 bytes (2 bytes); the first byte is the interrupt counter for the macros INTS_OFF and INTS_RES; the second byte contains the flags defined by using the macro FBIT

FLBIT - global, define label, [MUX; flag bit position assignment

counter for used with macro FBIT to define flags

FSET - global, macro, [MUX; macro to set the flag defined by using macro FBIT

FTST - global, macro, [MUX; macro to test the flag defined by using macro FBIT

FULL_WIR - global, equate, [MUX; transmission mode code for full duplex hardwired link

F_BIT - global, equate, [MUX; bit assignment for the WIC order request code specifying that the received buffer is to be flushed

GEN_NULL - global, equate, [MUX; bit assignment for UNIX_OPT, set to generate a null character instead of an event for the received break event

HANDSHAK - global, equate, [MUX; bit assignment for the special character tables P0SCHTBL, P1SCHTBL, ..., P7SCHTBL, set when the corresponding character is an handshake character

HAND_EN - global, equate, [MUX; bit assignment for RD_OPT, set when the handshake option is enabled

HEN_TCT - global, ALOC, [MUX, 2 bytes; the host ENQ/ACK or handshake timer counter, the first byte is millisecond down counter, and the second byte is the second down counter

HLED_ON - global, equate, [MUX; bit assignment for CAR_WREG, set when the hood LED is on

HOST_X_X - global, equate, [MUX; mask for the host X-ON/X-OFF bit in DEV_HAND

HOWTOSRQ - global, FBIT, [MUX; flag set when a data transfer occurred which tells subprogram SRQ_HOST to send a 10H to BIC register 5 to generate an SRQ; otherwise, send a 10H to BIC register 4 to generate an SRQ

HST_MASK - global, ALOC, [MUX, 1 byte; the first byte of the host interrupt mask, if the corresponding bit is set, the card will generate an event to the host if the card encountered the event

HST_MASL - global, ALOC, [MUX, 1 byte; the second byte of the host interrupt mask, same comment as for HST_MASK

HTIMEOUT - global, equate, [MUX; bit assignment for INT_STAT+1, set when the handshake timer timed out

HWDONE - global, FBIT, [MUX; flag set for the Z80 DMA MUX when the host write is doing a 1 byte transfer

HWD_XFER - global, FBIT, [MUX; flag set when the card is doing a host write data transfer

H_ACK - global, ALOC, [MUX, 1 byte; the host ENQ/ACK ACK character

H_D1_D3 - global, equate, [MUX; bit assignment for DEV_HAND, set when the host X-ON/X-OFF handshake is enabled

H_ENQ - global, ALOC, [MUX, 1 byte; the host ENQ/ACK ENQ character

H_EN_CTR - global, ALOC, [MUX, 1 byte; the host ENQ/ACK pacing counter value

H_EQ_AK - global, equate, [MUX; bit assignment for DEV_HAND, set when the host ENQ/ACK handshake is enabled

H_XOFF - global, ALOC, [MUX, 1 byte; contains the host X-ON/X-OFF X-OFF character

H_XON - global, ALOC, [MUX, 1 byte; contains the host X-ON/X-OFF X-ON character

IDY_BUF - local, code, &BCIDY, 9 bytes; contains the IDY information for the MUX card

IDY_LEN - local, equate, &BCIDY; the length of the MUX IDY block

IDY_RAM - local, variable, &BCIDY, 9 bytes; the RAM area for the IDY block, the MIC is unable to transfer data to the host from the ROM area, therefore the ROM data must first be moved to the RAM area

IFC - local, equate, &BPISR; bit assignment for BIC register 5, the interface clear bit

IM_XON - global, equate, [MUX; bit assignment for UNIX_OPT, set when the receiver is to restart the transmitter when any character is received while the transmitter is stopped due to an device X-OFF

INTS_CNT - global, equate, [MUX; equivalent to the first byte of FLAGS which is used as the interrupt counter, see also FLAGS

INT_STAT - global, ALOC, [MUX, 2 bytes; contains the interrupt

status

IOBUFLN - global, equate, [MUX; length of each I/O buffer

IOBUFRS - global, variable, &MUXVR, NPORTS*2*IOBUFLN bytes (8192 bytes); the I/O buffer for each port

IO_TABLE - local, code, &BCRWD, 12 bytes; jump table of the data transfer routines

IVECTOR - global, code, &MUXIV, 144 bytes; contains the interrupt service routine addresses, the receive interrupt cell addresses, the receive interrupt service routine addresses, and the speed sense interrupt service routine addresses

IV_RAM - global, equate, [MUX; the beginning RAM address for the interrupt table

IV_SIZE - global, equate, [MUX; the size of the interrupt table

LED_OFF - global, equate, [MUX; bit assignment for CAR_WREG, set when the card LED is off

LEN - local, code, &MXWCC, 4 bytes; the data mask for the number of significant data bits, index by the character length code

LF - global, equate, [MUX; the ASCII linefeed character

LINE_DEL - global, ALOC, [MUX, 1 byte; contains the character to be used as the line delete character for edit mode

MAXEV - global, equate, [MUX; maximum number of events per port

MAX_RX - global, equate, [MUX; maximum block size for the receive buffer

MAX_TX - global, equate, [MUX; maximum block size for the transmit buffer

MICVEC - global, equate, [MUX; the low byte of the interrupt vector address for the MIC

MIC_0 - local, equate, &BPISR; I/O port address for MIC register 0

MIC_1 - local, equate, &BPISR; I/O port address for MIC register 1

MIC_2 - local, equate, &BPISR; I/O port address for MIC register 2

MIC_3 - local, equate, &BPISR; I/O port address for MIC register 3

MIC_4 - local, equate, &BPISR; I/O port address for MIC register 4

MIC_5 - local, equate, &BPISR; I/O port address for MIC register 5

MIC_6 - local, equate, &BPISR; I/O port address for MIC register 6

MIC_7 - local, equate, &BPISR; I/O port address for MIC register 7

MIC_8 - local, equate, &BPISR; I/O port address for MIC register 8

MIC_9 - local, equate, &BPISR; I/O port address for MIC register 9

MIC_A - local, equate, &BPISR; I/O port address for MIC register 0AH

MIC_B - local, equate, &BPISR; I/O port address for MIC register 0BH

MIC_FLAG - global, define label, [MUXA; flag set to 0 when using Z80 DMA or to 1 when using MIC DMA. This flag is used in the conditional assembly statements in the following sources: &MUXMN, &BPISR, &MUXST, and &BCIDY.

MIC_IVEC - global, code, &MUXIV, 16 bytes; contains the MIC interrupt service routine addresses

MTERM - global, equate, [MUX; the offset within the RSR status block for the message termination character

MTYPE - global, equate, [MUX; the offset within the RSR status block for the message termination type code

MUX_BERR - global, equate, [MUX; the RSR error code for block mode is not allowed for the given request

MUX_CNEX - global, equate, [MUX; the RSR error code for cannot execute control request because no more space exist in the receive buffer

MUX_DAOV - global, equate, [MUX; the RSR error code for data overrun in the write data transfer

MUX_IREQ - global, equate, [MUX; the RSR error code for illegal request or request not implemented

MUX_IVSU - global, equate, [MUX; the RSR error code for illegal subfunction

MUX_NORX - global, equate, [MUX; the RSR error code for receive not allowed in simplex transmit mode

MUX_NOSP - global, equate, [MUX; the RSR error code for data transfer length too long

MUX_NOTX - global, equate, [MUX; the RSR error code for transmit not allowed in simplex receive mode

MUX_PARA - global, equate, [MUX; the RSR error code for illegal configuration parameter values

MUX_PLEN - global, equate, [MUX; the RSR error code for illegal configuration parameter length

MUX_PORT - global, equate, [MUX; the RSR error code for illegal port ID

NEEDTXSP - global, equate, [MUX; bit assignment for PORTSTAT+1, set when no more space exist in the transmit buffer to continue the host write request

NMK - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the NMI acknowledge bit

NO_PAR - global, equate, [MUX; bit assignment for UNIX_OPT, set when the "do not terminate receive record on errors" option is desired, the character in error will be replace by the replacement character found in RP_BAD

NPORTS - global, equate, [MUX; the number of ports on the card

OLDMASK1 - local, variable, &MXWCC, 1 byte; temporary storage to save the old content of HST_MASK while reconfiguring

OLDMASK2 - local, variable, &MXWCC, 1 byte; temproary storage to save the old content of HST_MASL while reconfiguring

ONE_ADDR - local, variable, &BPISR, 2 bytes; the buffer address for the last data byte in the host write request, used in the Z80 DMA MUX to process 1 data byte transfer and for fixing a BIC bug when both FFR and END condition are set at the same time

ORD - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the order bit

ORD_TBL - local, code, &BPISR, 32 bytes; jump table containing processing routine addresses corresponding each of the order being processed

OSEP_1 - global, ALOC, [MUX, 1 byte; contains the first output separator character

OSEP_2 - global, ALOC, [MUX, 1 byte; contains the second output separator character

OSEP_CT - global, ALOC, [MUX, 1 byte; specify the number character for the output separator

OTHER_RQ - global, ALOC, [MUX, 2 bytes; contains the address pointer to the other request block

P0ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 0 echo buffer

P0SCHTBL - global, variable, &MUXVR, 256 bytes; port 0 special character table

P0STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 0 port stuff information including port configuration, buffer pointers, counters, and so on

P1ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 1 echo buffer

P1SCHTBL - global, variable, &MUXVR, 256 bytes; port 1 special character table

P1STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 1 port stuff information including port configuration, buffer pointers, counters, and so on

P2ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 2 echo buffer

P2SCHTBL - global, variable, &MUXVR, 256 bytes; port 2 special character table

P2STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 2 port stuff information including port configuration, buffer pointers, counters, and so on

P3ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 3 echo buffer

P3SCHTBL - global, variable, &MUXVR, 256 bytes; port 3 special character table

P3STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 3 port stuff information including port configuration, buffer pointers, counters, and so on

P4ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 4 echo buffer

P4SCHTBL - global, variable, &MUXVR, 256 bytes; port 4 special character table

P4STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 4 port stuff information including port configuration, buffer pointers, counters, and so on

P5ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 5 echo buffer

P5SCHTBL - global, variable, &MUXVR, 256 bytes; port 5 special character table

P5STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 5 port stuff information including port configuration, buffer pointers, counters, and so on

P6ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 6 echo buffer

P6SCHTBL - global, variable, &MUXVR, 256 bytes; port 6 special character table

P6STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 6 port stuff information including port configuration, buffer pointers, counters, and so on

P7ECHO - global, variable, &MUXVR, EBLEN bytes (128 bytes); port 7 echo buffer

P7SCHTBL - global, variable, &MUXVR, 256 bytes; port 7 special character table

P7STUFF - global, variable, &MUXVR, P_MAPLEN bytes (185 bytes); port 7 port stuff information including port configuration, buffer pointers, counters, and so on

PAR - local, equate, &BPISR; bit assignment for BIC register 3, the perpherial address ready bit

PARITY - global, ALOC, [MUX, 1 byte; contains the parity option, 0 for no parity, 1 for odd, 2 for even, 3 for force '0', and 4 for force '1'

PER - global, equate, [MUX; BIC register 2 status code for protocol error

PID - global, ALOC, [MUX, 1 byte; use to get the port ID number from the WIC request block

PIDTABLE - global, variable, &MUXVR, NPORTS bytes (8 bytes); the

port ID table to map the logical port ID given in the WIC request to the physical port ID, the table is indexed by the physical port number

PORTABLE - local, code, &MUXMN, 16 bytes; table of addresses to the port stuff array, index by port number

PORTSTAT - global, ALOC, [MUX, 2 bytes; contains various flag for the port

PORT_DEF - local, code, &MUXMN, 60 bytes; the default port configuration

PORT_DFE - local, equate, &MUXMN; the ending address of PORT_DEF

PRIO_ABT - global, equate, [MUX; RTS op code priority for abort status

PRIO_BRK - global, equate, [MUX; RTS op code priority for break event

PRIO_CON - global, equate, [MUX; RTS op code priority for continue status

PRIO_END - global, equate, [MUX; RTS op code priority for end-of-data transfer

PRIO_ERR - global, equate, [MUX; RTS op code priority for error trap status

PRIO_IDL - global, equate, [MUX; RTS op code priority for nothing to do

PRIO_REC - global, equate, [MUX; RTS op code priority for data message received event

PRIO_SIG - global, equate, [MUX; RTS op code priority for signal character detected event

PRIO_SSM - global, equate, [MUX; RTS op code priority for speed sense status

PRIO_TIM - global, equate, [MUX; RTS op code priority for handshake timeout

PRIO_TX - global, equate, [MUX; RTS op code priority for transmit buffer is empty

P_MAPLEN - global, equate, [MUX; the length of port information block for each port

QS - local, equate, &MUXMN; the default quotable single text terminator

QUOTABLE - global, equate, [MUX; bit assignment for the special character table P0SCHTBL, P1SCHTBL, ..., P7SCHTBL, set for the corresponding character when it is quotable

QUOTE - global, ALOC, [MUX, 1 byte; the quotable character specified by the user

QUOT_MOD - global, equate, [MUX; bit assignment for UNIX_OPT, set when the quoting mode is enabled

Q_TEMP - global, variable, &MUXVR, 2 bytes; temporary storage for subprogram EDIT_Q parameters

RAM_END - global, equate, [MUX; the RAM ending address

RAM_STRT - global, equate, [MUX; the RAM starting address

RD_BUFR - global, ALOC, [MUX, 2 bytes; contains the buffer pointer to the next receive record for the backplane for the next read request; if no receive buffer is available, the pointer value will be zero

RD_BUF_N - global, ALOC, [MUX, 1 byte; the offset into the current backplane receive buffer for the next character

RD_CARD - global, equate, [MUX; the WIC request code for read card information

RD_DEV - global, equate, [MUX; the WIC request code for read device data

RD_OPT - global, ALOC, [MUX, 1 byte; contains the frontplane control options for processing the received data

RD_SUSP - global, equate, [MUX; bit assignment for PORTSTAT+1, set when the host read request is suspended

READ_DEC - local, equate, &BPISR; control word for MIC registers 3 and 8, read decrement configuration (ENO = stop, DIR = from, MEM = decr, ENI = off)

READ_INC - local, equate, &BPISR; control word for MIC registers 3 and 8, read increment configuration (ENO = stop, DIR = from, MEM = incr, ENI = off)

READ_RQ - global, ALOC, [MUX, 2 bytes; contains the address pointing the the read request block

REC_SEP - global, ALOC, [MUX, 1 byte; contains the record separator character to be used for searching for the conditional output separator option

REQ - global, ALOC, [MUX, 1 byte; index into the request block for the request code

RESUME - global, equate, [MUX; WIC op code to resume a transaction

RET_ADDR - local, variable, &BPISR, 2 bytes; contains the return address to the calling subprogram which requested a data transfer by calling H_CIR, H_READ, or H_WRITE

RFC - global, equate, [MUX; BIC register status code for ready for command

RITE_DEC - local, equate, &BPISR; control word for MIC registers 3 and 8, write decrement configuration (ENO = stop, DIR = to, MEM = decr, ENI = off)

RITE_INC - local, equate, &BPISR; control word for MIC registers 3 and 8, write increment configuration (ENO = stop, DIR = to, MEM = incr, ENI = off)

RLOG - global, equate, [MUX; offset into the status block for the residual count

RP_BAD - global, ALOC, [MUX, 1 byte; contains the replacement character to replace the receive character which contains an error. This option must be enabled by setting the NO_PAR bit in UNIX_OPT

RQA - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the request attention bit

RQB_LEN - global, equate, [MUX; request block length

RQ_FLIST - global, variable, &MUXVR, 2 bytes; link list of free request blocks

RQ_PSTUF - global, ALOC, [MUX, 2 bytes; index into the request block which contains the pointer to the port stuff for the request

RQ_TBLES - global, variable, [MUX, NPORTS*3*RQB_LEN bytes (480 bytes); allocate space for the request blocks

RSR_CODE - global, ALOC, [MUX, 1 byte; to contain the RSR status code

RSR_LEN - local, equate, &BCRSR; the length of the RSR block

RSR_RESD - global, ALOC, [MUX, 2 bytes; to contain the transmission residual count for the host read request

RSR_STT - global, ALOC, [MUX, 1 byte; to contain the single text terminator character for the host read request if the receive buffer is terminated by a single text terminator

RSR_TERM - global, ALOC, [MUX, 1 byte; to contain the text termination code for the host read request

RSR_TLOG - global, ALOC, [MUX, 2 bytes; to contain the transmission log for any data transfer between the host and the card

RSUB_STA - local, code, &MXRCI, 14 bytes; the jump table to the appropriate processing routine for the get status request (read card information)

RTSQ - global, variable, &MUXVR, 2 bytes; link list of the RTS response queue

RTS_ABT - global, equate, [MUX; the RTS status op code to abort the given transaction

RTS_CONT - global, equate, [MUX; the RTS status op code to continue the given transaction

RTS_END - global, equate, [MUX; the RTS status op code to terminate the data transfer phase of the given transaction

RTS_EROR - global, equate, [MUX; the RTS status op code to notify the host that the given transaction caused an error condition

RTS_EVNT - global, equate, [MUX; the RTS status op code to notify the host of an event (asynchronous interrupt)

RTS_IDLE - global, variable, &MUXVR, 1 byte; the RTS status op code to notify the host that the card has nothing to do, this value is not in ROM because the MIC will not DMA data from ROM to the host

RTS_LEN - global, equate, [MUX; the RTS block length portion of the event block

RTS_NONE - global, equate, [MUX; the RTS status op code for nothing to do (not used)

RTS_TEMP - global, variable, &MUXVR, 2 bytes; temporary storage area for subprogram BC_RTS

RWD_TABL - local, code, &BCRWD, 10 bytes; jump table to the

continuation processing routines after the data transfer

RXD_CTR - global, ALOC, [MUX, 1 byte; frontplane receive record down counter

RXD_CTRI - global, ALOC, [MUX, 1 byte; frontplane receive record down counter initial value

RXTX - local, equate, &BPISR; bit assignment for the I/O buffer address, clear for the receive buffer, set for the transmit buffer

RX_BFULL - global, equate, [MUX; bit assignment for RX_FLAGS, set when the receive buffer is full

RX_BUF_H - global, ALOC, [MUX, 2 bytes; frontplane active receive buffer header pointer

RX_DCTR - global, ALOC, [MUX, 2 bytes; end-on-count down counter

RX_FLAGS - global, ALOC, [MUX, 1 byte; receive configuration flags

RX_LCTR - global, ALOC, [MUX, 2 bytes; host read request down counter

RX_LOC - global, code, &MUXIV, 16 bytes; table of receive interrupt cell for each receive port

RX_NXT_C - global, ALOC, [MUX, 2 bytes; frontplane active receive buffer next character pointer

RX_SXOFF - global, equate, [MUX; bit assignment for PORTSTAT+1, set when the receiver have sent an X-OFF and is waiting for buffer space

RX_TABLE - local, code, &MUXMN, 8 bytes; table of high byte receive buffer address for each port, index by the port number

RX_TEMP - global, variable, &MUXVR, 2 bytes; temporary storage area for the receive interrupt service routine

RX_TXOFF - global, equate, [MUX; bit assignment for PORTSTAT, set when a device X-OFF is received and waiting for an X-ON

RX_VEC - global, code, &MUXIV, 16 bytes; a table of receive interrupt service routine address for each port

R_E_SENT - global, equate, [MUX; bit assignment for PORTSTAT+1, set when a message receive event was sent

SBIT_MSK - global, equate, [MUX; mask to get the S-bit from the WIC

request code

SCHARPTR - global, ALOC, [MUX, 1 byte; the high byte address of the special character table for the given port

SCHR_TBL - local, code, &MUXMN, 8 bytes; table containing the high byte address corresponding to each port special character table, index by the port number, this is used to initialize the port stuff array

SCR - global, equate, [MUX; BIC register 2 status code for subchannel connect request

SELF_TST - global, equate, [MUX; bit assignment for CAR_WREG, set when the card internal loopback mode is enabled for self-test

SEND_MES - global, equate, [MUX; bit assignment for DEV_HAND, set when the option to continue transmitting the message after the ENQ/ACK timer times out

SGENDDVR - global, equate, [MUX; bit assignment for CAR_WREG, set when the single-ended driver is enabled

SIGNAL - global, equate, [MUX; bit assignment in the special character table, set when the corresponding character is a signal character

SIGNALA - global, equate, [MUX; bit assignment for UNIX_OPT, set when the signal character detection option is enabled

SIGNAL_1 - global, ALOC, [MUX, 1 byte; contains the signal character 1

SIGNAL_2 - global, ALOC, [MUX, 1 byte; contains the signal character 2

SIGNAL_3 - global, ALOC, [MUX, 1 byte; contains the signal character 3

SIGNAL_4 - global, ALOC, [MUX, 1 byte; contains the signal character 4

SIG_1 - global, equate, [MUX; bit assignment for INT_STAT, set when the signal character 1 is detected

SIG_1_X - global, equate, [MUX; mask to get the signal character 1 value

SIG_2 - global, equate, [MUX; bit assignment for INT_STAT, set when the signal character 2 is detected

SIG_3 - global, equate, [MUX; bit assignment for INT_STAT, set when the signal character 3 is detected

SIG_4 - global, equate, [MUX; bit assignment for INT_STAT, set when the signal character 4 is detected

SIM_RX - global, equate, [MUX; the transmission code for simplex receive

SIM_TX - global, equate, [MUX; the transmission code for simplex transmit

SINGLE - global, equate, [MUX; bit assignment for RD_OPT, set when the frontplane control is to terminate the receive record when a single text terminator is encountered

SINGTEXT - global, equate, [MUX; bit assignment for the special character table, set when the corresponding character is to be used as a single text terminator

SIN_TEXT - global, ALOC, [MUX, 1 byte; the single text terminator character to cause the echoing of the CR-LF characters

SIOCMD - local, code, &MXCCD, 8 bytes; programming instructions to put the SIO in synchronous mode for speed sensing

SIOIVEC0 - global, code, &MUXIV, 16 bytes; the table of interrupt service routine addresses to service SIO #0

SIOIVEC1 - global, code, &MUXIV, 16 bytes; the table of interrupt service routine addresses to service SIO #1

SIOIVEC2 - global, code, &MUXIV, 16 bytes; the table of interrupt service routine addresses to service SIO #2

SIOIVEC3 - global, code, &MUXIV, 16 bytes; the table of interrupt service routine addresses to service SIO #3

SIOLEN - local, equate, &MXCCD; the size of SIOCMD

SIOVEC - global, equate, [MUX; the starting low byte address for the SIO interrupt vector table

SIO_0_AC - global, equate, [MUX; the I/O port address for the control channel of the SIO #0 channel A

SIO_0_AD - global, equate, [MUX; the I/O port address for the data channel of the SIO #0 channel A

SIO_0_BC - global, equate, [MUX; the I/O port address for the control channel of the SIO #0 channel B

SIO_0_BD - global, equate, [MUX; the I/O port address for the data channel of the SIO #0 channel B

SIO_1_AC - global, equate, [MUX; the I/O port address for the control channel of the SIO #1 channel A

SIO_1_AD - global, equate, [MUX; the I/O port address for the data channel of the SIO #1 channel A

SIO_1_BC - global, equate, [MUX; the I/O port address for the control channel of the SIO #1 channel B

SIO_1_BD - global, equate, [MUX; the I/O port address for the data channel of the SIO #1 channel B

SIO_2_AC - global, equate, [MUX; the I/O port address for the control channel of the SIO #2 channel A

SIO_2_AD - global, equate, [MUX; the I/O port address for the data channel of the SIO #2 channel A

SIO_2_BC - global, equate, [MUX; the I/O port address for the control channel of the SIO #2 channel B

SIO_2_BD - global, equate, [MUX; the I/O port address for the data channel of the SIO #2 channel B

SIO_3_AC - global, equate, [MUX; the I/O port address for the control channel of the SIO #3 channel A

SIO_3_AD - global, equate, [MUX; the I/O port address for the data channel of the SIO #3 channel A

SIO_3_BC - global, equate, [MUX; the I/O port address for the control channel of the SIO #3 channel B

SIO_3_BD - global, equate, [MUX; the I/O port address for the data channel of the SIO #3 channel B

SIO_BRK - global, equate, [MUX; bit assignment for SIO channel B read register 0 indicating the break receive condition

SIO_CS - global, equate, [MUX; bit assignment for SIO read register 0 indicating the state of the clear-to-send modem signal

SIO_CTRL - global, ALOC, [MUX, 1 byte; contains the SIO I/O port address of the control channel for the port

SIO_DEF - local, code, &MUXMN, 8 bytes; contains the programming instructions for initializing the SIO

SIO_EXEN - global, equate, [MUX; bit assignment for SIO write register 1 to enable the SIO external status interrupt

SIO_FRER - global, equate, [MUX; bit assignment for SIO read register 1 for the framing error bit

SIO_LEN - global, equate, &MUXVR; the length of the SIO programming instructions

SIO_PAR - global, equate, [MUX; bit assignment for SIO read register 1 for the parity error bit

SIO_RR - global, equate, [MUX; bit assignment for SIO read register 0 indicating the state of the receiver ready modem signal

SIO_RS - global, equate, [MUX; bit assignment for SIO write register 5 to set the request-to-send modem signal state

SIO_RXAV - global, equate, [MUX; bit assignment for SIO read register 0 indicating whether a receive character is available

SIO_RXEN - global, equate, [MUX; bit assignment for SIO write register 3 to enable or disalbe the receive interrupt

SIO_RXIN - global, equate, [MUX; bit assignment for SIO write register 3 to set the receive interrupt mode to interrupt on all receive characters

SIO_RXOV - global, equate, [MUX; bit assignment for SIO read register 1 for the data overrun error bit

SIO_SNBK - global, equate, [MUX; bit assignment for SIO write register 5 to enable and disable the break generator

SIO_TBL - local, code, &MUXMN, 8 bytes; table of SIO I/O port address for the control channel for each port, index by the port number

SIO_TR - global, equate, [MUX; bit assignment for SiO write register 5 to set or clear the terminal ready modem signal

SIO_TXEM - global, equate, [MUX; bit assignment for SIO read register 0 indicating whether the transmit buffer is empty

SIO_TXEN - global, equate, [MUX; bit assignment for SIO write register 1 to enable the transmitter

SIO_TXIN - global, equate, [MUX; bit assignment for SIO write register 1 to enable the transmit interrupt

SIO_W - global, variable, &MUXVR, 8 bytes; array of SIO programming instructions

SIO_W3 - global, variable, &MUXVR, 1 byte; contains the SIO write register 3 content

SIO_W4 - global, variable, &MUXVR, 1 byte; contains the SIO write register 4 content

SIO_W5 - global, variable, &MUXVR, 1 byte; contains the SIO write register 5 content

SPACE - global, equate, [MUX; the ASCII space character

SQUOTE - global, ALOC, [MUX, 1 byte; the single text terminator to be used as a quoting character when quoting mode is enabled in UNIX

SRE - local, equate, &BPISR; bit assignment for BIC registers 5 and 6, the status register is empty bit

SS_CHAR - global, ALOC, [MUX, 1 byte; contains the previous character processing in speed sense mode

SS_CTR - global, ALOC, [MUX, 10 bytes; 5 16-bit counter for speed sensing

SS_DCTR - global, ALOC, [MUX, 1 byte; the down counter for the number of remaining speed sense counter for counting

SS_PTR - global, ALOC, [MUX, 2 bytes; the address pointer to the current counter in SS_CTR

SS_VEC - global, code, &MUXIV, 16 bytes; the table of speed sense interrupt service routine addresses for each port

ST1 - local, equate, &MUXMN; the default single text terminator

STATUS - global, equate, &MUXVR; equivalent to FLAGS since no status bits were defined

STERM - global, ALOC, [MUX, 8 bytes; an array of single text terminators

STERM_CT - global, ALOC, [MUX, 1 byte; the number of valid single text terminators in array STERM

STOP_BIT - global, ALOC, [MUX, 1 byte; specify the stop bit option, 0 for 1, 1 for 1.5, and 2 for 2

STP - local, equate, &BPISR; bit assignment for BIC register 3, the

self-test pass bit

STRP_MSK - global, ALOC, [MUX, 1 byte; contains the stripping option information

STRP_TRM - global, equate, [MUX; bit assignment for STRP_MSK, set the text terminators are to be stripped from the receive buffer

ST_CT - local, equate, &MUXMN; the default number of single text terminator

ST_TEMP - global, ALOC, [MUX, 9 bytes; contains the new single text terminator information which will be enabled after the current frontplane record is terminated. If the first byte is zero, no new single text terminator configuration is available.

SUBFCN - global, ALOC, [MUX, 1 byte; the subfucntion code of the WIC request

SUB_CON - global, FBIT, [MUX; flag set when the subchannel is connected

SUB_PSE - global, FBIT, [MUX; flag set when the subchannel is paused

S_BIT - global, equate, [MUX; bit assignment for the WIC request code for the S-bit which specify whether to keep the partial buffer (if any) for the next read

TEMP - global, variable, &MUXVR, 2 bytes; temporary storage

TEMP_BUF - local, variable, &MXWDD, 2 bytes; buffer address for the next data transfer

TEMP_LEN - local, variable, &MXWDD, 2 bytes; buffer length for the next data transfer

TERM_AL1 - global, equate, [MUX; termination code in the receive buffer header for alert 1

TERM_BOF - global, equate, [MUX; termination code in the receive buffer header for receive buffer overflow

TERM_BUF - global, equate, [MUX; termination code in the receive buffer header for terminated by card, user buffer full

TERM_CNT - global, equate, [MUX; termination code in the receive buffer header for termination on count

TERM_FER - global, equate, [MUX; termination code in the receive buffer header for framing error

TERM_HOS - global, equate, [MUX; termination code in the receive buffer header for termination initiated by the host

TERM_OVF - global, equate, [MUX; termination code in the receive buffer header for SIO data overrun error

TERM_PER - global, equate, [MUX; termination code in the receive buffer header for parity error

TERM_PRT - global, equate, [MUX; termination code in the receive buffer header for termination by card, partial record

TERM_STT - global, equate, [MUX; termination code in the receive buffer header for single text termination

TID - global, ALOC, [MUX, 2 bytes; the transaction ID number from the WIC request block

TIMR_TYP - global, equate, [MUX; bit assignment for STRP_MSK, set to specify an event to be generated when the handshake timer timed out

TLOG - global, equate, [MUX; offset into the RSR status block for the transmission log

TOGL_MSK - global, ALOC, [MUX, 1 byte; contains the frontplane features toggle mask

TR_ADDR - global, ALOC, [MUX, 2 bytes; contains the data transfer address for the transaction

TR_LEN - global, ALOC, [MUX, 2 bytes; contains the data transfer length for the transaction

TX_BUSY - global, equate, [MUX; bit assignment for PORTSTAT, set when the transmitter is busy transmitting data

TX_CTR - global, ALOC, [MUX, 1 byte; the frontplane transmitter down counter

TX_DATA - global, equate, [MUX; bit assignment for PORTSTAT, set when the transmitter is busy transmitting user data and not echo data

TX_ECHO - global, equate, [MUX; bit assignment for PORTSTAT+1, set when the echo buffer is not empty

TX_EMPTY - global, equate, [MUX; bit assignment for INT_STAT, set

when the transmit buffer is empty

TX_ENQ - global, equate, [MUX; bit assignment for PORTSTAT, set when an ENQ character is to be transmitted on the next transmit interrupt for the port

TX_HXOFF - global, equate, [MUX; bit assignment for PORTSTAT, set when the transmitter needs to transmit the host X-OFF character on the next transmit interrupt

TX_HXON - global, equate, [MUX; bit assignment for PORTSTAT, set when the transmitter needs to transmit the host X-ON character on the next transmit interrupt

TX_MODE - global, ALOC, [MUX, 1 byte; contains the transmission mode option, 2 for full duplex hardwired, 3 for simplex transmit, and 4 for simplex receive

TX_OUT2 - global, equate, [MUX; bit assignment for PORTSTAT, set when the second character of the output separator needs to be transmitted for the conditional output separator option

TX_PTR - global, ALOC, [MUX, 2 bytes; contains the address pointing to the next character for data transmission

TX_TABLE - local, code, &MUXMN, 8 bytes; table of the high byte transmit buffer address for each port, index by the port number

T_D1_D3 - global, equate, [MUX; bit assignment for DEV_HAND, set when the device X-ON/X-OFF handshake is enabled

T_EQ_AK - global, equate, [MUX; bit assignment for DEV_HAND, set when the device ENQ/ACK handshake is enabled

T_XOFF - global, ALOC, [MUX, 1 byte; device X-OFF character

T_XON - global, ALOC, [MUX, 1 byte; device X-ON character

UNIX_OPT - global, ALOC, [MUX, 1 byte; contains the miscellaneous flags to implement some of the UNIX features

WAIT_ACK - global, equate, [MUX; bit assignment for PORTSTAT, set when the firmware is waiting for the ACK character for the ENQ character sent

WAPPEND - global, equate, [MUX; bit assignment for the subfunction code in the WIC write device data request, set to append the output separators to the message being written

WHAND - global, equate, [MUX; bit assignment for the subfunction code in the WIC write device data request, set to toggle the handshake option

WCC_LEN - global, code, &MXWCC, 35 bytes; a table containing the length for each corresponding subfunction parameters. If the entry contains a zero, then the length is variable and is checked in the code. If the entry contains a 0FFH value, the corresponding subfunction is not used by this firmware. Otherwise, the nonzero value is the length of the parameters that must be given by the user.

WCC_TAB - global, code, &MXWCC, 105 bytes; a table containing the offset from the start of the port stuff array for the subfunction parameter and the processing routine address to validate the parameters

WIC_TEMP - global, variable, &MUXVR, 2 bytes; temporary storage area for subprogram BC_WIC

WRITE_RQ - global, ALOC, [MUX, 2 bytes; contains the address pointing to the write request block

WR_CARD - global, equate, [MUX; WIC request code for write card configuration

WR_DEV - global, equate, [MUX; WIC request code for write device data

WSUB_MX - local, equate, &MXWCC; the number of valid subfunction code for WCC request and the size of WCC_LEN

WTC_BUFR - global, variable, &MUXVR, 3 bytes; buffer area for the WTC order request block

WTC_LEN - global, equate, [MUX; WTC request block length

WTC_OPCD - global, ALOC, [MUX, 1 byte; contains the WTC op code

WTC_PID - global, ALOC, [MUX, 1 byte; contains the port ID in the WTC request block

WTC_TID - global, ALOC, [MUX, 2 bytes; contains the transaction ID in the WTC request block

Z80DMA - global, equate, [MUX; the Z80 DMA I/O port address

SUBPROGRAM & JUMP ENTRY SYMBOLS — CHAPTER 3

The following is description for each subprogram and jump entry symbols used in the MUX firmware. There are many multiple entries in each module to reduce the memory space required. This may cause a little difficulty in understanding and modifying the firmware, but the primary objective of fitting the firmware into an 16K EPROM was met.

The symbol is listed first followed by the source file where it is defined. The linkage and the subprogram called are given next, respectively, if the symbol represent a subprogram which can be CALL. Following this is the list of all subprograms calling or jumping to this symbol. Finally a short functional description is given.

ABRT_REQ - source: &BCRSR

linkage: CALL ABRT_REQ

Register IY contains the address of the request block.

calls: FREE_RQB, NOTHIN

jump to: none

called by: BIC_RSR, BIC_WTC

used by: none

Subprogram to kill off a request and deallocate its resource.

ADD_Q - source: &MUXEV

linkage: CALL ADD_Q, register DE contains the address of the queue header, register HL contains the address of the block

calls: SRQ_HOST

jump to: none

called by: BIC_RTS, EAK_IT

used by: none

Routine to add an event block to the RTS or event queues. Blocks are queued in priority order (0 = highest).

AL1_EVNT - source: &RXISR

linkage: CALL AL1_EVNT

On entry register IY = address of port's stuff.

calls: EVNT_MGR, GET_EVB

jump to: none

called by: RDD_CONT, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Subprogram to generate an alert-1 event to the host.

ARQ_HOST - source: &BPISR

linkage: CALL ARQ_HOST

On entry register C contains the ARQ status code.

calls: none

jump to: none

called by: BIC_ABT, BIC_ERR, BIC_SC, DIE

used by: none

The subprogram to send the ARQ status code to the host through the BIC register 2. If the BIC register is busy, queue the status until the host is ready for it.

BIC_ABT - source: &BPISR

linkage: JP BIC_ABT

calls: ARQ_HOST

jump to: none

called by: none

used by: BIC_ISR

The BIC abort command processor.

BIC_DIS - source: &BPISR

linkage: JP BIC_DIS

calls: none

jump to: BIC_EXIT

called by: none

used by: BIC_ISR

BIC disconnect order processor.

BIC_ERR - source: &BIC_ERR

linkage: JP BIC_ERR

calls: ARQ_HOST

jump to: BIC_EXIT

called by: none

used by: BIC_ISR, BIC_RD, BIC_WD, BIC_WIC, BIC_WTC

Send the protocol error status to the host.

BIC_EXIT - source: &BPISR

linkage: JP BIC_EXIT

calls: none

jump to: none

called by: none

used by: BIC_DIS, BIC_END, BIC_ERR, BIC_PSE, BIC_SC, DMAB_ISR, HCIR_IO, HRD_IO, HWD_IO, ORD_EXIT

SRE_RTN

Restore the registers and return from BIC or MIC interrupt.

BIC_IDY - source: &BCIDY

linkage: CALL BIC_IDY

calls: HRD_IO, SRQ_HOST

jump to: BIC_EXIT

called by: BIC_ISR

used by: none

The MUX IDY order handler. Move the IDY information to RAM, and then send it to the host. The information is moved to RAM first, because the MIC cannot DMA out of the ROM area.

BIC_INIT - source: &BPISR

linkage: CALL BIC_INIT

calls: none

jump to: none

called by: MUX_MAIN

used by: none

The subprogram to initialize the BIC ISR, the BIC, and the MIC.

BIC_ISR - source: &BPISR

linkage: CALL BIC_ISR

calls: ARQ_HOST, BIC_IDY, BIC_RD, BIC_RSR, BIC_RTS, BIC_WD, BIC_WIC, BIC_WTC

jump to: BIC_EXIT

called by: BIC interrupt

used by: MIC_IVEC

BIC and MIC interrupt service routine.

BIC_PSE - source: &BPISR

linkage: JP BIC_PSE

calls: SRQ_HOST

jump to: BIC_EXIT

called by: none

used by: BIC_ISR

BIC PAUSE order handler.

BIC_RES - source: &BPISR

linkage: JP BIC_RES

calls: SRQ_HOST

jump to: none

called by: none

used by: BIC_ISR

BIC RESUME command processor.

BIC_RD - source: &BCRWD

linkage: CALL BIC_RD

calls: BIC_ERR, HCIR_IO HRD_IO, HWD_IO

jump to: ORD_EXIT, RCI_CONT, RDD_CONT, WCC_CONT, WDD_CONT

called by: BIC_ISR

used by: none

RD and WD order handler.

BIC_RSR - source: &BCRSR

linkage: CALL BIC_RSR

calls: ABRT_REQ, HRD_IO

jump to: ORD_EXIT

called by: BIC_ISR

used by: none

This subprogram returns the read request status block for the current transaction to the host.

BIC_RTS - source: &BCRTS

linkage: CALL BIC_RTS

calls: ADD_Q, FREE_EVB, GET_EVB, HRD_IO, NOTHIN, SRQ_HOST, UPDTID

jump to: BIC_EXIT

called by: BIC_ISR

used by: none

This subprogram handles all RTS order processing.

BIC_SC - source: &BPISR

linkage: JP BIC_SC

calls: ARQ_HOST, SRQ_HOST

jump to: BIC_EXIT

called by: none

used by: BIC_ISR

BIC subchannel connect command handler.

BIC_WD - same as BIC_RD

BIC_WIC - source: &BCWIC

linkage: CALL BIC_WIC

calls: CCD_BEG, CDV_BEG, EVNT_MGR, GET_EVB, GET_RQB, HWD_IO, RCI_BEG, RDD_BEG, SRQ_HOST, WCC_BEG, WDD_BEG

jump to: BIC_EXIT, DIE

called by: BIC_ISR

used by: none

WIC order handler.

BIC_WTC - source: &BCWTC

linkage: CALL BIC_WTC

calls: ABRT_REQ, ADD_Q, CCD_ABT, CDV_ABT, FREE_EVB, HWD_IO, RCI_ABT RDD_ABT, SET_EVNT, WCC_ABT, WDD_ABT, WDD_END

jump to: BIC_ERR, ORD_EXIT

called by: BIC_ISR

used by: none

WTC order handler.

CCD_ABT - source: &MXCCD

linkage: CALL CCD_ABT

calls: none

jump to: none

called by: BIC_WTC

used by: none

Abort the control card transaction.

CCD_BEG - source: &MXCCD

linkage: CALL CCD_BEG

On entry register BC contains the port stuff address, register IY contains the request block

address.

On exit the C flag is set to continue the transaction, the S flag is set if the transaction contains an error, the Z flag is set to suspend the transaction (not used).

calls: CHUCK_RX, DATA_TX, ECHO_CK, MSG_EVNT, PACKITUP, PRG_CTC, RESYNC, RX_COMPL, SEND_XON, SET_CNTR, SET_EVNT, SET_SIO

jump to: none

called by: BIC_WIC

used by: none

Control card request processor.

CDV_ABT - source: &MXCDV

linkage: CALL CDV_ABT

calls: none

jump to: none

called by: BIC_WTC

used by: none

Abort the control device request processor.

CDV_BEG - source: &MXCDV

linkage: CALL CDV_BEG

calls: none

jump to: none

called by: BIC_WIC

used by: none

Control device request processor.

CHEK_XOF - source: &RXISR

linkage: CALL CHEK_XOF

On entry register IY = address of port's stuff.

calls: PUT_CHAR, SET_CNTR, SP_LEFT

jump to: none

called by: RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, SPC_ISR0, SPC_ISR1, SPC_ISR2, SPC_ISR3, SPC_ISR4, SPC_ISR5, SPC_ISR6, SPC_ISR7

used by: none

The frontplane down counter has hit zero; check to see if it is time to send an X-OFF to the device. If so (and it is enabled) send the character. Unless called from RX_COUNT subprogram, the frontplane down counter is updated.

CHUCK_RX - source: &MXCCD

linkage: CALL CHUCK_RX

Register IY contains the port stuff address.

calls: EDIT_IT

jump to: none

called by: CCD_BEG

used by: none

Clean up the receive event.

CTCIVEC0 - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

The CTC #0 interrupt vector table.

DATA_CK - source: &TXISR

linkage: CALL DATA_CK

On entry register C = SIO data channel I/O port address, register D = parity mask, register E = data mask, register IY = port stuff address.

calls: none

jump to: DATA_TXC

called by: none

used by: ECHO_CK

Subprogram to set up pointers & counters to start data transmission.

DATA_CKA - source: &TXISR

linkage: JP DATA_CKA

On entry registers C, D, E, and IY must be set up the same as for DATA_CK. In addition the buffer address must be set in register HL.

calls: none

jump to: none

called by: none

used by: PUT_DATA

Same function as for DATA_CK except register HL is already set and it bypass some of the checking which is unnecessary.

DATA_TX - source: &TXISR

linkage: CALL DATA_TX

On entry register C = SIO data channel I/O port address, register D = parity mask, register E = data mask, register IY = port stuff address.

calls: SET_EVNT, TX_SP_CK

jump to: none

called by: DATA_TX, TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

Subprogram to send the next character, update the buffer pointer and counter.

DATA_TXC - source: &TXISR

linkage: JP DATA_TXC

On entry the content of registers C, D, E, and IY are the same as for entry into DATA_TX except register HL must contain the buffer pointer for the next output character.

calls: see DATA_TX

jump to: none

called by: none

used by: DATA_CK

Same function as DATA_TX except register HL has the buffer pointer already and need not be set up again.

DIE - source: &BPISR

linkage: JP DIE

calls: ARQ_HOST

jump to: none

called by: none

used by: GET_EVB, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7

Subprogram to send the dead-or-dying status error to the host and then jump to 0 to reinitialize the card.

DMAA_ISR - source: &DMAA

linkage: CALL DMAA_ISR

calls: none

jump to: none

called by: MIC channel A DMA interrupt

used by: MIC_IVEC

This subprogram keep track of DMA channel A interrupts which should never occur, except under bad condition.

DMAB_ISR - source: &BPISR

linkage: CALL DMAB_ISR

calls: none

jump to: BIC_EXIT, return to the subprogram which calls HCIR_IO, HRD_IO, or HWD_IO

called by: MIC interrupt

used by: MIC_IVEC

MIC channel B DMA interrupt service routine.

EAK_IT - source: &BCWTC

linkage: CALL EAK_IT

calls: ADD_Q, GET_STUF

jump to: none

called by: BIC_WTC, EDIT_IT

used by: none

Subprogram to process the EAK request of the WTC order.

ECHO_CK - source: &TXISR

linkage: CALL ECHO_CK

On entry register C = SIO data channel I/O port address, register D = parity mask, register E = data mask, register IY = port stuff address.

calls: HENQ_CK, PUT_CHAR

jump to: DATA_CK

called by: ECHO_CK, TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

Subprogram to check echo buffer and transmit next echo character. If echo buffer is empty, go check the user buffer.

EDIT_IT - source: &MUXEV

linkage: CALL EDIT_IT, (Q_TEMP) = the event to be edited

calls: EAK_IT, EDIT_Q

jump to: none

called by: CHUCK_RX, RCI_BEG, RDD_BEG, WCC_CONT, WDD_BEG

used by: none

Routine to do all the editing, busy work, etc. to remove an event from the system.

EDIT_Q - source: &MUXEV

linkage: CALL EDIT_Q, (Q_TEMP) = event code, (Q_TEMP+1) = port ID, register DE = address of the queue header, interrupt system must be off

calls: FREE_EVB

jump to: none

called by: EDIT_IT

used by: none

Routine to remove events from either the RTS queue or the event queue.

EVNT_MGR - source: &MUXEV

linkage: CALL EVNT_MGR, register HL contains the address of the RTS or event queue

All registers will be destroyed except for register IY.

calls: ADD_Q

jump to: none

called by: AL1_EVNT, BIC_WIC, GENSEVEN, MSG_EVNT, SET_EVNT

used by: none

Routine to manage the queue of events and RTS responses.

EXT_ISR0 - source: &EXISR

linkage: CALL EXT_ISR0

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #0 channel A (port 0)

used by: SIOIVEC0

Process the external status interrupt generated by the SIO #0 channel A. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR1 - source: &EXISR

linkage: CALL EXT_ISR1

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #0 channel B (port 1)

used by: SIOIVEC0

Process the external status interrupt generated by the SIO #0 channel B. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR2 - source: &EXISR

linkage: CALL EXT_ISR2

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #1 channel A (port 2)

used by: SIOIVEC1

Process the external status interrupt generated by the SIO #1 channel A. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR3 - source: &EXISR

linkage: CALL EXT_ISR3

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #1 channel B (port 3)

used by: SIOIVEC1

Process the external status interrupt generated by the SIO #1 channel B. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR4 - source: &EXISR

linkage: CALL EXT_ISR4

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #2 channel A (port 4)

used by: SIOIVEC2

Process the external status interrupt generated by the SIO #2 channel A. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR5 - source: &EXISR

linkage: CALL EXT_ISR5

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #2 channel B (port 5)

used by: SIOIVEC2

Process the external status interrupt generated by the SIO #2 channel B. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR6 - source: &EXISR

linkage: CALL EXT_ISR6

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #3 channel A

used by: SIOIVEC3

Process the external status interrupt generated by the SIO #3 channel A. The ISR only process the break condition. All other conditions are ignored.

EXT_ISR7 - source: &EXISR

linkage: CALL EXT_ISR7

calls: SET_EVNT

jump to: none

called by: external status interrupt on SIO #3 channel B

used by: SIOIVEC3

Process the external status interrupt generated by the SIO #3 channel B. The ISR only process the break condition. All other conditions are ignored.

FIND_TID - source: &BCWTC

linkage: CALL FIND_TID

On exit the Z flag is set if the subprogram cannot find the request block containing the TID. Otherwise, register DE contains the request block address.

calls: none

jump to: none

called by: BIC_WIC

used by: none

Subprogram to find the location of the request block containing the TID specified in the WTC request block.

FP_W1000 - source: &MXWCC

linkage: JP FP_W1000

calls: none

jump to: none

called by: none

used by: RCI_BEG

Entry in WCC_BEG to return the request cannot be block error.

FP_W1060 - source: &MXWCC

linkage: JP FP_W1060

calls: none

jump to: none

called by: none

used by: RCI_BEG

Entry in WCC_BEG to return the invalid subfunction code error.

FP_W1200 - source: &MXWCC

linkage: JP FP_W1200

calls: none

jump to: none

called by: none

used by: RCI_BEG

Entry in WCC_BEG to return the illegal request length error.

FREE_EVB - source: &MUXEV

linkage: CALL FREE_EVB, register HL contains the block address

calls: none

jump to: none

called by: BIC_RTS, BIC_WTC, EDIT_Q, MUX_MAIN

used by: none

Routine to return an event block to the free space storage area.

FREE_RQB - source: &BCWIC

linkage: CALL FREE_RQB

On entry register H1 contains the address of the block to be returned to free storage.

calls: none

jump to: none

called by: BIC_RSR, MUX_MAIN

used by: none

Routine to return a request block to free storage.

GENSEVEN - source: &SSISR

linkage: CALL GENSEVEN

Register B contains the baud rate index sensed, register C contains the 8th bit value of character sensed, register IY contains the port stuff address.

calls: EVNT_MGR, GET_EVB

jump to: none

called by: SS_ISR0, SS_ISR1, SS_ISR2, SS_ISR3, SS_ISR4, SS_ISR5, SS_ISR6, SS_ISR7

used by: none

Generate the solicited speed sensed event.

GET_EVB - source: &MUXEV

linkage: CALL GET_EVB

Returns the address to the block in register HL. Register A is trashed.

calls: none

jump to: DIE

called by: AL1_EVNT, BIC_RTS, BIC_WIC, GENSEVEN, MSG_EVNT, SET_EVNT

used by: none

Allocate an event block from free storage.

GET_RQB - source: &BCWIC

linkage: CALL GET_RQB

On exit the Z flag bit is set if there is no space. Otherwise register HL will contain the address of the block

calls: none

jump to: none

called by: BIC_WIC

used by: none

Subprogram to get a request block from the free list.

GET_STUF - source: &BCWTC

linkage: CALL GET_STUF

On entry, register A contains the port ID.

On exit, register HL contains the address of PORTSTAT+1 for the port being processed.

calls: none

jump to: none

called by: EAK_IT

used by: none

A subprogram to find the address for PORTSTAT+1 for the port specified in the WTC request block.

HCIR_IO - source: &BPISR

linkage: CALL HCIR_IO

Register DE contains the data transfer length and register HL contains the buffer address.

calls: none

jump to: none

called by: BIC_RD, BIC_WD

used by: none

This subprogram is to transfer data from or to the host to the card transmit buffer or from the card receive buffer which are structured as circular buffers.

HENQ_CK - source: &TXISR

linkage: CALL HENQ_CK

calls: none

jump to: none

called by: ECHO_CK, PUT_ECHO, TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

Check and process the host ENQ/ACK handshake option.

HRD_IO - source: &BPISR

linkage: CALL HRD_IO

Register DE contains the data transfer length and register HL contains the buffer address.

calls: none

jump to: none

called by: BIC_IDY, BIC_RD, BIC_RSR, BIC_RTS, BIC_WD

used by: none

Subprogram to transfer data from the card to the host. The subprogram will automatically assert the end on the last byte.

HWD_IO - source: &BPISR

linkage: CALL HWD_IO

calls: none

jump to: none

called by: BIC_RD, BIC_WD, BIC_WIC, BIC_WTC

used by: none

Subprogram to transfer data from the host to the card.

IVECTOR - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

The MUX interrupt table.

MIC_IVEC - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

The MIC interrupt vector table.

MIN - source: &RXISR

linkage: CALL MIN

On entry register HL = one value, register DE = the other value.

On exit register HL contains the smaller value.

calls: none

jump to: none

called by: RDD_BEG, RX_COMPL, SET_CNTR

used by: none

Subprogram to return the minimum of the two 16-bit values.

MSG_EVNT - source: &MXRDD

linkage: CALL MSG_EVNT

Register DE = address of the buffer, register IY = address of port's stuff.

calls: EVNT_MGR, GET_EVB

jump to: none

called by: CCD_BEG, RDD_COMPL, RDD_CONT, WCC_CONT

used by: none

Subprogram to generate a message received event block from the current backplane receive buffer.

MUX_DDM - source: &MXDDM

linkage: JP MUX_DDM

calls: none

jump to: none

called by: none

used by: MUX_MAIN

The debug monitor for trouble shooting the MUX product firmware. The only way to enter is by forcing an NMI on the Z80 by touch pin 17 to ground.

The debug monitor is programmed to transmit and receive data only on port 0. The baud rate is 9600. Additional documentation may be found in the source listing.

MUX_MAIN - source: &MUXMN

linkage: main firmware entry always start at address zero

calls: BIC_INIT, FREE_EVB, FREE_RQB, OTSP_SET, SING_SET

jump to: MX_STEST

called by: main entry

used by: none

Main firmware entry to initialize system.

MX_STEST - source: &MUXST

linkage: JP MX_STEST

Register IY contains the return address.

calls: none

jump to: the caller

called by: none

used by: MUX_MAIN

The MUX card self-test.

NOTHIN - source: &BCRSR

linkage: CALL NOTHIN

calls: none

jump to: none

called by: ABRT_REQ, BIC_RTS, RDD_CONT

used by: none

Subprogram to make the active request idle.

ORD_EXIT - source: &BCRTS

linkage: JP ORD_EXIT

calls: SRQ_HOST

jump to: BIC_EXIT

used by: BIC_RD, BIC_RSR, BIC_RTS, BIC_WD, BIC_WTC

Subprogram to do the SRQ for the next order and to jump to the BIC exit point.

OTSP_CLR - source: &MXWCC

linkage: CALL OTSP_CLR

Register IY contains the port stuff address.

calls: none

jump to: none

called by: WCC_CONT

used by: none

Subprogram to clear all the special condition in the special character table except for the single text termination.

OTSP_SET - source: &MXWCC

linkage: CALL OTSP_SET

Register IY contains the port stuff address.

calls: none

jump to: none

called by: MUX_MAIN, WCC_CONT

used by: none

Subprogram to set all the special condition in the special character table except for the single text termination.

PACKITUP - source: &RXISR

linkage: CALL PACKITUP

On entry register A = terminating reason, register E = terminating character, and register IY = address of port's stuff.

On exit register HL = address of buffer's

header.

calls: none

jump to: none

called by: CCD_BEG, RDD_BEG, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Subprogram to pack up the current receive buffer, sets the length and the terminating reason in the buffer header, and sets the pointers for the next buffer.

PRG_CTC - source: &MXWCC

linkage: CALL PRG_CTC

On entry register IY contains the address to the port stuff.

calls: none

jump to: none

called by: CCD_BEG, SS_ISR0, SS_ISR1, SS_ISR2, SS_ISR3, SS_ISR4, SS_ISR5, SS_ISR6, SS_ISR7, WCC_CONT

used by: none

Program the baud rate generator for the specified port.

PUT_CHAR - source: &TXISR

linkage: CALL PUT_CHAR

On entry register A contains the character for output, registr C contains the SIO data channel I/O port address, register IY contains the port stuff address.

calls: none

jump to: none

called by: CHEK_XOF, ECHO_CK, PUT_ECHO, RDD_BEG, REAL_CLK, SEND_XON, TX_ISR0, TX_ISR1,

TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

Check for force parity and then output the next character.

PUT_CHR - source: &RXISR

linkage: CALL PUT_CHR

On entry register IY = address of port's stuff, register E = character to store.

On exit, the Z flag is set on buffer full.

calls: none

jump to: none

called by: RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, SPC_ISR0, SPC_ISR1, SPC_ISR2, SPC_ISR3, SPC_ISR4, SPC_ISR5, SPC_ISR6, SPC_ISR7

used by: none

Subprogram to place a character into the active receive buffer and update all the pointers.

PUT_DATA - source: &TXISR

linkage: CALL PUT_DATA

On entry register IY contains the port stuff address.

calls: none

jump to: DATA_CKA

called by: WDD_BEG, WDD_CONT

used by: none

Subprogram to start the data transmission, if necessary.

PUT_ECHO - source: &TXISR

linkage: CALL PUT_ECHO

On entry register A contains the character to be echo, register IY contains the port stuff address.

calls: HENQ_CK, PUT_CHAR

jump to: none

called by: RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, SPC_ISR0, SPC_ISR1, SPC_ISR2, SPC_ISR3, SPC_ISR4, SPC_ISR5, SPC_ISR6, SPC_ISR7

used by: none

Subprogram to output an echo character if the transmitter is not busy. Otherwise, add the character to the echo buffer.

RCI_ABT - source: &MXRCI

linkage: CALL RCI_ABT

calls: none

jump to: none

called by: BIC_WTC

used by: none

Abort the read card information transaction.

RCI_BEG - source: &MXRCI

linkage: CALL RCI_BEG

On entry register BC = port stuff address, register IY = request block address.

On exit the C flag is set to continue, the S flag is set for an error, the Z flag is set to do nothing.

calls: EDIT_IT

jump to: FP_W1000, FP_W1060, FP_W1200

called by: BIC_WIC

used by: none

Begin read card information processing. Set up the buffer address and the buffer length for the transaction.

RCI_CONT - source: &MXRCI

linkage: CALL RCI_CONT

calls: none

jump to: none

called by: BIC_RD

used by: none

Continue the read card information transaction.

RDD_ABT - source: &MXRDD

linkage: CALL RDD_ABT

On entry register BC contains the port stuff address, register IY contains the request block address.

calls: RX_TOGGL

jump to: none

called by: BIC_WTC

used by: none

Abort the read device data transaction.

RDD_BEG - source: &MXRDD

linkage: CALL RDD_BEG

On entry register BC contains the port stuff address, register IY contains the request block address.

On exit the C flag is set to continue the transaction, the S flag is set if the transaction contains an error, the Z flag is set to suspend the transaction.

calls: EDIT_IT, MIN, PACKITUP, PUT_CHAR, RD_SWAP, RX_TOGGL, SET_CNTR, UPD_EOC

jump to: none

called by: BIC_WIC

used by: none

Begin read device data transaction processing.

RDD_CONT - source: &MXRDD

linkage: CALL RDD_CONT

On entry register BC contains the port stuff address, register IY contains the request block address.

calls: AL1_EVNT, MSG_EVNT, NOTHIN, SEND_XON, SET_CNTR, UPD_EOC

jump to: none

called by: BIC_RD

used by: none

Continue the read device data transaction.

RD_SWAP - source: &MXWCC

linkage: CALL RD_SWAP

Register IY = port stuff address.

calls: SING_CLR, SING_SET

jump to: none

called by: RDD_BEG, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Make the pending read configuration active.

REAL_812 - source: &MXCLK

linkage: JP REAL_812

Register IY contains the port stuff address.

calls: none

jump to: TX_90, TX_280

called by: none

used by:

An entry into REAL_CLK to set up the environment to jump to the transmit ISR.

REAL_CLK - source: &MXCLK

linkage: CALL REAL_CLK

calls: PUT_CHAR, SET_EVNT

jump to: TX_90, TX_280

called by: CTC #0 channel 3 interrupt

used by: CTCIVEC0

Real time clock interrupt service routine. The clock resolution is 10 milliseconds.

RESYNC - source: &MXCCD

linkage: CALL RESYNC

Register IY contains the port stuff address.

calls: none

jump to: none

called by: CCD_BEG, SS_ISR0, SS_ISR1, SS_ISR2, SS_ISR3, SS_ISR4, SS_ISR5, SS_ISR6, SS_ISR7

used by: none

Subprogram to put the SIO into synchronous mode and the corresponding CTC to 19.2 KHz to start speed sensing.

RX_COMPL - source: &MXRDD

linkage: CALL RX_COMPL

Register HL = address of completed buffer, register IY = address of port stuff, the interrupt system must be off before calling.

calls: MIN, MSG_EVNT, SET_CNTR, SET_EVNT

jump to: none

called by: CCD_BEG, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Subprogram to complete a read request to the host. The current buffer is made available to the backplane, and either an event is generated if no request is pending, or a continue status is generated if there is a request.

RX_IS0A - source: &RXISR

linkage: JP RX_IS0A

calls: see RX_ISR0

jump to: see RX_ISR0

called by: none

used by: SPC_ISR0

Entry in RX_ISR0 to continue the receive character processing.

RX_IS1A - source: &RXISR

linkage: JP RX_IS1A

calls: see RX_ISR1

jump to: see RX_ISR1

called by: none

used by: SPC_ISR1

Entry in RX_ISR1 to continue the receive character processing.

RX_IS2A - source: &RXISR

linkage: JP RX_IS2A

calls: see RX_ISR2

jump to: see RX_ISR2

called by: none

used by: SPC_ISR2

Entry in RX_ISR2 to continue the receive character processing.

RX_IS3A - source: &RXISR

linkage: JP RX_IS3A

calls: see RX_ISR3

jump to: see RX_ISR3

called by: none

used by: SPC_ISR3

Entry in RX_ISR3 to continue the receive character processing.

RX_IS4A - source: &RXISR

linkage: JP RX_IS4A

calls: see RX_ISR4

jump to: see RX_ISR4

called by: none

used by: SPC_ISR4

Entry in RX_ISR4 to continue the receive character processing.

RX_IS5A - source: &RXISR

linkage: JP RX_IS5A

calls: see RX_ISR5

jump to: see RX_ISR5

called by: none

used by: SPC_ISR5

Entry in RX_ISR5 to continue the receive character processing.

RX_IS6A - source: &RXISR

linkage: JP RX_IS6A

calls: see RX_ISR6

jump to: see RX_ISR6

called by: none

used by: SPC_ISR6

Entry in RX_ISR6 to continue the receive character processing.

RX_IS7A - source: &RXISR

linkage: JP RX_IS7A

calls: see RX_ISR7

jump to: see RX_ISR7

called by: none

used by: SPC_ISR7

Entry in RX_ISR7 to continue the receive character processing.

RX_LOC - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

Contains the location of the receive interrupt cell for the RX interrupt service routine address.

RX_ISR0 - source: &RXISR

linkage: CALL RX_ISR0

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #0 channel A receive interrupt

used by: RX_VEC, SIOIVEC0

Process the character received from the SIO.

RX_ISR1 - source: &RXISR

linkage: CALL RX_ISR1

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #0 channel B receive interrupt

used by: RX_VEC, SIOIVEC0

Process the character received from the SIO.

RX_ISR2 - source: &RXISR

linkage: CALL RX_ISR2

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #1 channel A receive interrupt

used by: RX_VEC, SIOIVEC1

Process the character received from the SIO.

RX_ISR3 - source: &RXISR

linkage: CALL RX_ISR3

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #1 channel B receive interrupt

used by: RX_VEC, SIOIVEC0

Process the character received from the SIO.

RX_ISR4 - source: &RXISR

linkage: CALL RX_ISR4

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #2 channel A receive interrupt

used by: RX_VEC, SIOIVEC2

Process the character received from the SIO.

RX_ISR5 - source: &RXISR

linkage: CALL RX_ISR5

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #2 channel B receive interrupt

used by: RX_VEC, SIOIVEC2

Process the character received from the SIO.

RX_ISR6 - source: &RXISR

linkage: CALL RX_ISR6

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #3 channel A receive interrupt

used by: RX_VEC, SIOIVEC3

Process the character received from the SIO.

RX_ISR7 - source: &RXISR

linkage: CALL RX_ISR7

calls: AL1_EVNT, CHEK_XOF, PACKITUP, PUT_CHR, PUT_ECHO, RX_COMPL, RD_SWAP, SET_CNTR, SET_EVNT, SP_LEFT

jump to: DIE

called by: SIO #3 channel B receive interrupt

used by: RX_VEC, SIOIVEC3

Process the character received from the SIO.

RX_SPAC - source: &MXWCC

linkage: CALL RX_SPAC

On entry register IY contains the port stuff address.

On exit register HL contains the result, register A contains the content in register L.

calls: none

jump to: none

called by: RCI_BEG, WCC_CONT

used by: none

Find the current frontplane receive record size which also includes th header.

RX_TOGGL - source: &MXRDD

linkage: CALL RX_TOGGL

Register BC contains the address of the request block, register IY contains the address of port stuff.

calls: none

jump to: none

called by: RDD_BEG

used by: none

Routine to set up the special character function mask for the frontplane. Each bit in the read request subfunction is used to toggle the enabled/disabled state of certain frontplane functions: handshake, signal, quotable, edit, and single text termination. This routine also toggles the echo mode flag in RX_FLAGS.

RX_VEC - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

Contains the normal receive interrupt service routine addresses.

SEND_XON - source: &MXCCD

linkage: CALL SEND_XON

calls: PUT_CHAR, SP_LEFT

jump to: none

called by: CCD_BEG, RDD_CONT

used by: none

Subprogram to test if an X-OFF have been sent by the receiver. If so, and there is now enough space (due to flush) send an X-ON and clear the X-OFF sent flag.

SET_CNTR - source: &RXISR

linkage: CALL SET_CNTR

On entry register IY contains the address to port's stuff.

calls: MIN

jump to: none

called by: CCD_BEG, RDD_BEG, RDD_CONT, RX_COMPL, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Subprogram to set the frontplane down counter.

SET_DMSK - source: &MXWCC

linkage: CALL SET_DMSK

Register IY contains the port stuff address.

calls: none

jump to: none

called by: WCC_CONT

used by: none

Set the data and parity mask.

SET_EOC - source: &MXWCC

linkage: CALL SET_EOC

Register IY contains the port stuff address.

calls: none

jump to: none

called by: WCC_CONT

used by: none

Routine to copy the master value of the end-on-count counter to the active (running) frontplane counter.

SET_EVNT - source: &MUXEV

linkage: CALL SET_EVNT

Register C contains the event priority, register B contains the RTS status code (CONTINUE, END, EVENT SENSED). If B = CONTINUE or END, register DE contains the address of the request block. If B = EVENT SENSED, then register D = event code (e.g., TX buffer available) and register E = port number.

calls: EVNT_MGR, GET_EVB

jump to: none

called by: BIC_WTC, CCD_BEG, DATA_TX, EXT_ISR0, EXT_ISR1, EXT_ISR2, EXT_ISR3, EXT_ISR4, EXT_ISR5, EXT_ISR6, EXT_ISR7, REAL_CLK, RX_COMPL, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3, RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, WCC_CONT

used by: none

Subprogram to interface the frontplane to the backplane event manager.

SET_SIO - source: &MXWCC

linkage: CALL SET_SIO

Register IY = address of port stuff.

calls: none

jump to: none

called by: CCD_BEG, SS_ISR0, SS_ISR1, SS_ISR2, SS_ISR3, SS_ISR4, SS_ISR5, SS_ISR6, SS_ISR7, WCC_CONT

used by: none

Configure the SIO for the specified port.

SING_CLR - source: &MXWCC

linkage: CALL SING_CLR

Register IY contains the address to port stuff, the character to be cleared is in array STERM.

calls: none

jump to: none

called by: WCC_CONT

used by: none

Clear the single text termination condition in the special character table.

SING_SET - source: &MXWCC

linkage: CALL SING_SET

Register IY contains the address to port stuff, the single text terminator to be set is in array STERM.

calls: none

jump to: none

called by: MUX_MAIN, WCC_CONT

used by: none

Set the single text termination condition bit in the special character table for the given character.

SIOIVEC0 - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

SIO #0 interrupt vector table.

SIOIVEC1 - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

SIO #1 interrupt vector table.

SIOIVEC2 - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

SIO #2 interrupt vector table.

SIOIVEC3 - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

SIO #3 interrupt vector table.

SPC_ISR0 - source: &SPISR

linkage: CALL SPC_ISR0

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS0A

called by: receive special condition SIO #0 channel A interrupt

used by: RX_VEC, SIOIVEC0

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR1 - source: &SPISR

linkage: CALL SPC_ISR1

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS1A

called by: receive special condition SIO #0 channel B interrupt

used by: RX_VEC, SIOIVEC0

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR2 - source: &SPISR

linkage: CALL SPC_ISR2

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS2A

called by: receive special condition SIO #1 channel B interrupt

used by: RX_VEC, SIOIVEC1

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR3 - source: &SPISR

linkage: CALL SPC_ISR3

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS3A

called by: receive special condition SIO #1 channel B interrupt

used by: RX_VEC, SIOIVEC1

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR4 - source: &SPISR

linkage: CALL SPC_ISR4

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS4A

called by: receive special condition SIO #2 channel A interrupt

used by: RX_VEC, SIOIVEC2

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR5 - source: &SPISR

linkage: CALL SPC_ISR5

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS5A

called by: receive special condition SIO #2 channel B interrupt

used by: RX_VEC, SIOIVEC2

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR6 - source: &SPISR

linkage: CALL SPC_ISR6

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS6A

called by: receive special condition SIO #3 channel A interrupt

used by: RX_VEC, SIOIVEC3

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the framing error generated by the SIO.

SPC_ISR7 - source: &SPISR

linkage: CALL SPC_ISR7

calls: CHEK_OF, PUT_CHR, PUT_ECHO

jump to: RX_IS7A

called by: receive special condition SIO #3 channel B interrupt

used by: RX_VEC, SIOIVEC3

Receive special condition interrupt service routine. This ISR handles the parity, data overrun, and the

framing error generated by the SIO.

SPD_SEN - source: &SPDSN

linkage: CALL SPD_SEN

On entry register A contains the new character received, register B contains the previous character, register C contains the down counter, register DE is used as a scratch, register HL contains the address to the current counter for the port. The first counter in RAM must be initialize at the time the SIO is put into hunt mode.

On exit the C flag is set to continue speed sensing, the S flag is set if speed sense failed, the Z flag is set if the speed sense routine found the correct baud rate. In the latter case registers A and B will contains the index into the baud rate table given in subprogram PRG_CTC.

calls: none

jump to: none

called by: SS_ISR0, SS_ISR1, SS_ISR2, SS_ISR3, SS_ISR4, SS_ISR5, SS_ISR6, SS_ISR7

used by: none

Speed sense subprogram to detect the baud rate of the selected port.

SP_LEFT - source: &MXRDD

linkage: CALL SP_LEFT

On entry register IY = port stuff address.

On exit register HL contains the space remaining.

calls: none

jump to: none

called by: CHEK_XOF, RX_ISR0, RX_ISR1, RX_ISR2, RX_ISR3,

RX_ISR4, RX_ISR5, RX_ISR6, RX_ISR7, SEND_XON

used by: none

Subprogram to figure out how much space is left in the receive buffer.

SRE_RTN - source: &BPISR

linkage: JP SRE_RTN

calls: none

jump to: BIC_EXIT

called by: none

used by: BIC_ISR

Subprogram to send the next ARQ status code when the BIC is ready for it.

SRQ_HOST - source: &BPISR

linkage: CALL SRQ_HOST

calls: none

jump to: none

called by: ADD_Q, BIC_IDY, BIC_PSE, BIC_RSE, BIC_RTS, BIC_SC, BIC_WIC

used by: none

Send the SRQ to the host for the next order.

SSB_ISR0 - source: &SSBIR

linkage: CALL SSB_ISR0

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR1 - source: &SSBIR

linkage: CALL SSB_ISR1

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR2 - source: &SSBIR

linkage: CALL SSB_ISR2

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR3 - source: &SSBIR

linkage: CALL SSB_ISR3

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR4 - source: &SSBIR

linkage: CALL SSB_ISR4

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR5 - source: &SSBIR

linkage: CALL SSB_ISR5

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR6 - source: &SSBIR

linkage: CALL SSB_ISR6

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SSB_ISR7 - source: &SSBIR

linkage: CALL SSB_ISR7

calls: RESYNC

jump to: none

called by: none

used by: SS_VEC

Reinitialize speed sensing due to a data overrun error.

SS_ISR0 - source: &SSISR

linkage: CALL SS_ISR0

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #0 channel A interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR1 - source: &SSISR

linkage: CALL SS_ISR1

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #0 channel B interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR2 - source: &SSISR

linkage: CALL SS_ISR2

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #1 channel A interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR3 - source: &SSISR

linkage: CALL SS_ISR3

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #1 channel B interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR4 - source: &SSISR

linkage: CALL SS_ISR4

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #2 channel A interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR5 - source: &SSISR

linkage: CALL SS_ISR5

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #2 channel B interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR6 - source: &SSISR

linkage: CALL SS_ISR6

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #3 channel A interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_ISR7 - source: &SSISR

linkage: CALL SS_ISR7

calls: GENSEVEN, PRG_CTC, RESYNC, SET_SIO, SPD_SEN

jump to: none

called by: receive SIO #3 channel B interrupt when in speed sensing mode

used by: SS_VEC

Speed sensing interrupt service routine.

SS_VEC - source: &MUXIV

linkage: none

calls: none

jump to: none

called by: none

used by: none

Speed sense interrupt service routine address table.

TOGGLE - source: &MXWDD

linkage: CALL TOGGLE

On entry register BC contains the port stuff address, register IY contains the request block address.

calls: none

jump to: none

called by: WDD_BEG, WDD_CONT

used by: none

Toggle the handshake bit in the toggle mask.

TOGGLEA - source: &MXWCC

linkage: CALL TOGGLEA

Register IY contains the address to port stuff, the interrupt system must be off before calling.

calls: none

jump to: none

called by: WCC_CONT

used by: none

Update the toggle mask due to subfunction 0, 1, or 31 changes.

TX_280 - source: &TXISR

linkage: JP TX_280

calls: see TX_ISRx

jump to: none

called by: none

used by: REAL_812, REAL_CLK

Continuation in the transmitter ISR to decide whether to continue transmission of user data or echo data.

TX_90 - source: &TXISR

linkage: JP TX_90

calls: see TX_ISRx

jump to: none

called by: none

used by: REAL_812, REAL_CLK

Entry into the TX ISR to check for transmitter options and to continue data transmission.

TX_EQAK - source: &TXISR

linkage: CALL TX_EQAK

calls: none

jump to: none

called by: TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

Subprogram to do the ENQ/ACK counter processing when the counter goes to zero.

TX_ISR0 - source: &TXISR

linkage: CALL TX_ISR0

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #0 channel A interrupt

used by: SIOIVEC0

SIO transmit interrupt service routine.

TX_ISR1 - source: &TXISR

linkage: CALL TX_ISR1

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK,

TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #0 channel B interrupt

used by: SIOIVEC0

SIO transmit interrupt service routine.

TX_ISR2 - source: &TXISR

linkage: CALL TX_ISR2

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #1 channel A interrupt

used by: SIOIVEC1

SIO transmit interrupt service routine.

TX_ISR3 - source: &TXISR

linkage: CALL TX_ISR0

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #1 channel B interrupt

used by: SIOIVEC1

SIO transmit interrupt service routine.

TX_ISR4 - source: &TXISR

linkage: CALL TX_ISR4

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #2 channel A interrupt

used by: SIOIVEC2

SIO transmit interrupt service routine.

TX_ISR5 - source: &TXISR

linkage: CALL TX_ISR5

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #2 channel B interrupt

used by: SIOIVEC2

SIO transmit interrupt service routine.

TX_ISR6 - source: &TXISR

linkage: CALL TX_ISR6

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #3 channel A interrupt

used by: SIOIVEC3

SIO transmit interrupt service routine.

TX_ISR7 - source: &TXISR

linkage: CALL TX_ISR7

calls: DATA_TX, ECHO_CK, HENQ_CK, PUT_CHAR, TX_EQAK, TX_OUTSP, TX_PACK

jump to: none

called by: TX SIO #3 channel B interrupt

used by: SIOIVEC3

SIO transmit interrupt service routine.

TX_OUTSP - source: &TXISR

linkage: CALL TX_OUTSP

calls: see DATA_TX

jump to: none

called by: TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

An entry into DATA_TX at the point where it start to check for conditional output separators.

TX_PACK - source: &TXISR

linkage: CALL TX_PACK

calls: see DATA_TX

jump to: none

called by: TX_ISR0, TX_ISR1, TX_ISR2, TX_ISR3, TX_ISR4, TX_ISR5, TX_ISR6, TX_ISR7

used by: none

An entry into DATA_TX to do the transmit buffer termination after all the data has been transmitted.

TX_SP_CK - source: &MXWDD

linkage: CALL TX_SP_CK

On entry register DE contains the required length for the data transfer, register IY contains the port stuff address.

On exit the S flag is set if there is no space.

calls: none

jump to: none

called by: DATA_TX, WDD_BEG, WDD_CONT

used by: none

Check to see if have enough space for the write device data request.

UPD_EOC - source: &MXRDD

linkage: CALL UPD_EOC

On entry register IY contains the port stuff address.

On exit register HL contains the new counter value.

calls: none

jump to: none

called by: RDD_BEG, RDD_CONT, WCC_CONT

used by: none

Routine to update the end-on-count running counter in preparation for calling SET_CNTR in the middle of a receive record.

WCC_ABT - source: &MXWCC

linkage: CALL WCC_ABT

On entry register BC = port stuff pointer, register IY = request block pointer.

calls: none

jump to: none

called by: BIC_WTC

used by: none

Abort the write card configuration transaction.

WCC_BEG - source: &MXWCC

linkage: CALL WCC_BEG

On entry register BC contains the address of port stuff, register IY contains the address of the request block.

On exit the C flag is set to continue the transaction, the S flag is set if the transaction contains an error, the Z flag is set to suspend the transaction.

calls: none

jump to: none

called by: BIC_WIC

used by: none

Begin the write card configuration transaction.

WCC_CONT - source: &MXWCC

linkage: CALL WCC_CONT

On entry register BC contains the port stuff address, register IY contains the request block address, ACTSTAT contains the RTS_CONT code.

On exit set ACTSTAT to its new RTS status code, if necessary.

calls: AL1_EVNT, EDIT_IT, MSG_EVNT, OTSP_CLR, OTSP_SET, PACKITUP, PRG_CTC, RD_SWAP, RX_COMPL, RX_SPAC, SET_CNTR, SET_DMSK, SET_EOC, SET_EVNT, SET_SIO, SING_CLR, SING_SET, TOGGLEA, UPD_EOC

jump to: none

called by: BIC_WD

used by: none

Write card configuration transaction continuation/completion processor.

WDD_ABT - source: &MXWDD

linkage: CALL WDD_ABT

On entry register BC contains the port stuff address, register IY contains the request block address.

calls: TOGGLE

jump to: none

called by: BIC_WTC

used by: none

Abort the write device data transaction.

WDD_BEG - source: &MXWDD

linkage: CALL WDD_BEG

On entry register BC contains the port stuff address, register IY contains the request block address.

On exit the C flag is set to continue the transaction, the S flag is set if the transaction contains an error, the Z flag is set to suspend the transaction.

calls: EDIT_IT, PUT_DATA, TOGGLE, TX_SP_CK

jump to:none

called by: BIC_WIC

used by: none

Begin the write device data transaction.

WDD_CONT - source: &MXWDD

linkage: CALL WDD_CONT

On entry register BC contains the port stuff address, register IY contains the request block address, ACTSTAT contains the RTS_CONT status code.

On exit set ACTSTAT to its new RTS status code, if necessary

calls: PUT_DATA, TOGGLE, TX_SP_CK

jump to: none

called by: BIC_WD

used by: none

Continue or complete the write device data transaction.

WDD_END - source: &MXWDD

linkage: CALL WDD_END

On entry register BC contains the port stuff address, register IY contains the request block address.

calls: TOGGLE

jump to: none

called by: BIC_WTC

used by: none

Terminate the write device data transaction.

RECEIVE BUFFER MANAGEMENT	CHAPTER 4

The receive buffer management is performed by the receive ISR (RXISR) and the receive midplane processor MXRDD. Some manipulations are done by control card (MXCCD) for buffer flush.

Receive buffers originate on the frontplane, set up as the last step of PACKITUP when terminating the previous record. There are two frontplane pointers, RX_BUF_H which points to the active buffer's header, and RX_NXT_C which points to where the next character will be stored. The backplane has a pointer to the active backplane receive buffer, and an offset to where the next character to be read into the host will be fetched from. The pointers are 16 bit values, the offset 8 bits.

The buffer's header consists of three bytes, one for length (including the header itself), one for terminating conditions and errors, and one for the terminating character. The header is initially cleared, and is filled in when the buffer is terminated (except for errors which or-in to the second byte).

Receive buffers are managed in a circular fashion within the 512 byte receive buffer space per port. The space used is that which is between the backplane buffer header address and the next character pointer for the frontplane. There is always space at the end for a zero header, which is guaranteed by SET_CNTR's algorithm for setting the frontplane down counter, and RX_COUNT's setting of the RX_BFULL flag.

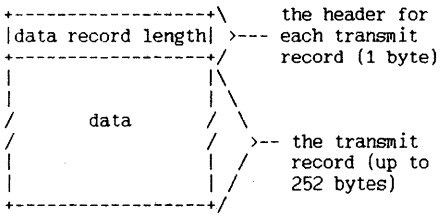
When a record is terminated by PACKITUP the current frontplane header is filled in, and the frontplane pointers are moved to point to the next buffer. The old pointer to the header is given to the backplane if it currently does not know about one (pointer is zero). If the backplane already knows about a record there is nothing to do to give this new one to it. When the records ahead of the newly terminated one are read up to the host by MXRDD the backplane's pointers are automatically moved until a zero header is detected. The zero header is, of course, the current frontplane record.

TRANSMIT BUFFER MANAGEMENT	CHAPTER 5

The transmit buffer is organized as a circular buffer consisting of 512 bytes. This allows the card to buffered many data blocks from the host before suspending the read request due to no buffer space.

Each data block in the transmit buffer consists of a 1-byte header followed by the transmit data. The data block size can vary from 2 to 253 bytes.

A Transmit Data Block in the Circular Buffer



- The data block length is equal to the sum of the header length and the data record length.
- The data record length is the number of data bytes in the transmit data record.
- The header is always preallocated for the next host write device data request before the current record is made available for the frontplane.
- The data record length is set by the backplane subprogram WDD_CONT when the buffer is ready for transmission. The backplane also calls the frontplane subprogram PUT_DATA to start the transmitter if necessary.
- If there is not enough space in the transmit buffer to hold the write device data request plus the space for the next header plus the space for the two output separators plus the space for the next header of the next record, the write request will be suspended. The frontplane subprogram DATA_TX

will restart the transaction when space becomes available.

- The backplane buffer pointer IY+BPTX_PTR points to the beginning of the current block to contain the next transmit data record. This pointer is updated after each write order in the transaction.
- The frontplane buffer pointer IY+TX_PTR points to the next character in the transmit buffer for transmission.
- The frontplane transmit down counter, IY+TX_CTR, specify the amount of data remaining in the current record for transmission. When this counter reaches zero, the buffer pointer IY+TX_PTR will be pointing to the header of the next data block.
- IY is the base register containing the address to port stuff for the port being processed. IY is set before entering its respective processing subprograms.
- The amount of remaining space in the transmit buffer is computed as follow:

  (IY+TX_PTR) - (IY+BPTX_PTR) + 1

  If the result is less than 0, add 512.

- The transmit buffer is defined in RAM as follow:

  port 0 - C200 to C3FF

  port 1 - C600 to C7FF

  port 2 - CA00 to CBFF

  port 3 - CE00 to CFFF

  port 4 - D200 to D3FF

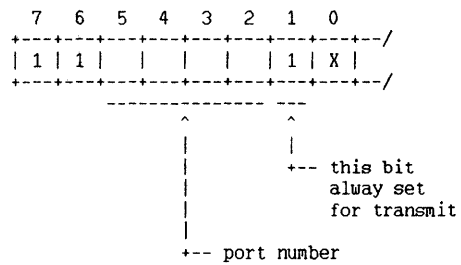
  port 5 - D600 to D7FF

  port 6 - DA00 to DBFF

  port 7 - DE00 to DFFF

  This information is used in managing the buffers. For example, to update the buffer pointer, bit 1 is clear, the pointer is incremented, and then the bit 1 is set again. This will take care of the wraparound of the circular buffer.

high byte portion of address to TX buffer



ECHO BUFFER MANAGEMENT — CHAPTER 6

- The echo buffer is managed as a 128 bytes circular buffer.
- The buffer is managed by using two pointers, IY+ECHOPTRI and IY+ECHOPTRO, where IY is the base register containing the address to the "port stuff" array corresponding to the port being accessed.
- The pointer ECHOPTRI points to the next location in the echo buffer for the next character to be stored.
- The pointer ECHOPTRO points to the next character in the buffer for data transmission.
- If the pointers ECHOPTRI and ECHOPTRO points to the same location in the buffer, then the echo buffer is empty. This is true even if the buffer overflow. There is no check for buffer overflow.
- The echo buffer address will be from XX00 to XX7F or from XX80 to XXFF depending upon whether the port is even or odd, respectively.

  For even ports (0, 2, 4, and 6), the firmware checks the low byte of the address to see if it is 7F for a wraparound. If it is true, the low byte of the address is reset to 0.

  Likewise for ports 1, 3, 5, and 7, the firmware checks the low byte of the address to see if it is FF for a wraparound. If it is true, the low byte of the address is reset to 80.

- This buffer is managed by subprograms ECHO_CK and PUT_ECHO.

ARQ BUFFER MANAGEMENT	CHAPTER 7

- The ARQ buffer is managed as a first-in first-out (FIFO) queue.
- There is no check for buffer overflow. This should never happen.
- The buffer is managed by using two pointers, ARQ_PTRA and ARQ_PTRB. ARQ_PTRA is the write pointer, and ARQ_PTRB is the read pointer. When a status is to be added to the queue, put the status into the ARQ buffer by using the address in ARQ_PTRA. Then increment the content to the next address. If the address goes beyond the buffer, reset the pointer to the beginning of the ARQ buffer.

  When the status register is empty on the BIC, read the status code found in the pointer ARQ_PTRB. Then increment the content to the next address. If the address goes beyond the buffer, reset the pointer to the beginning of the ARQ buffer.

  When the two pointers are equal, then the ARQ buffer is empty.
- This buffer is managed by subprograms ARQ_HOST and SRE_RTN.

SOFTWARE HANDSHAKE ALGORITHMS	CHAPTER 8

The following are the algorithms implemented for the software handshake

8.1 Host ENQ/ACK Handshake

This option is used to pace the data transfer from the card to the device to prevent the device from losing any data due to its slow internal processing speed.

The firmware will send the ENQ character in H_ENQ after its pacing counter ENQ_DCTR counts down to zero. In addition the flag TX_HENQ in PxSTUFF will be set, the counter will be reset to the value from H_EN_CTR which is programmable by the user, and finally the host ENQ/ACK timer HEN_TCT will be set with the value from ENQ_TIMR, if it is active.

The firmware will not transmit any data until an host ACK is received from the device, or when the host ENQ/ACK timer times out and the transmit on ENQ timer time out option is enabled. However, the transmitter will send an host ENQ again if the timer times out and the transmit on ENQ timer time out option is disabled.

Subprogram HENQ_CK will decrement the count and set the flag TX_ENQ in PORTSTAT or subprogram TX_ISRx will decrement the count and subprogram TX_EQAK will set the flag. On the next transmit interrupt the flag TX_ENQ will be checked. If it is set, the ENQ character from H_ENQ will be sent. The timer ENQ/ACK or handshake timer HEN_TCT will be set with the value from ENQ_TIMR. The flag TX_ENQ will be cleared at this time and the flag WAIT_ACK in PORTSTAT will be set to wait for the ACK.

Subprogram REAL_CLK perform the timer time out operation; and subprogram RX_ISRx checks for the incoming ACK.

Note that the ENQ/ACK counter is always be decremented in the transmitter whenever a character is transmitted. When the counter goes to zero, the firmware then check to see whether to process the handshake. This is faster then checking for handshake enable first and then decrementing the counter.

8.2 Host X-ON/X-OFF Handshake

This handshake protocol allows the card to pace the data transfer from the device to the card. The card will send the X-OFF character in H_OFF to the device to stop data transmission when there is not enough space in the receive buffer. This character is sent by calling PUT_ECHO when the transmitter is not busy. If the transmitter is busy, the character is sent by setting the flag TX_HXOFF for subprogram TX_ISRx to send it on the next interrupt.

The receive ISR is told when to send an XOFF by the value set in the frontplane down counter by SET_CNTR. When this decrements to zero, the space remaining in the port's receive buffer is checked. If it is less than 72 bytes, an XOFF condition is indicated, and the character is sent.

Receive space and XOFF condition set is checked in MXCCD subfunctions 1 and 2 (buffer flush), and MXRDD when a record is read into the host. If there is sufficient space created, an XON is sent to the terminal. Due to the length of the buffer header (3 bytes) plus the data in the buffer (at least 1 byte), there is a minimum of 4 bytes of histerisis in the XON / XOFF handshake.

8.3 Device X-ON/X-OFF Handshake

This handshake protocol allows the device to pace the data transfer from the card to the device. The device will signal the card to stop transmitting data by sending an X-OFF character. The firmware will set the flag RX_TXOFF to tell the transmitter to stop transmitting.

Data transmission may be resumed by the device sending an X-ON character or by typing any characters if the implicit X-ON option is enabled. The user also have the option of restarting the transmitter by using the control card request with subfunction 5.

This device handshake is also invoked using control card request subfunction 8 to suspend the transmitter. Note that the handshake option does not have to be enabled to use this control request. The transmitter may be restarted by the same methods as described above, except that the control card request must be used if the handshake is disabled.

TIMER ALGORITHMS — CHAPTER 9

The card uses CTC #0 channel 3 as the firmware real time clock. The clock resolution is set to 10 millisecond.

The second timer is implemented by using 2 bytes to perform 2 count down. The first counter is used to count the 10's millisecond time down to the "second" resolution. Whenever the first counter counts down to zero, the second counter is decremented by 1 to count down the seconds.

The handshake timers uses this algorithm for the host ENQ/ACK and device X-ON/X-OFF handshakes. The host ENQ/ACK timer is used to determine when to send another host ENQ character or to restart the transmitter. If the timer times out and if the restart the transmitter option on time-out is disabled, another host ENQ character will be sent to the device and the counter will be set to time out again. If the restart the transmitter option is enabled, the host ENQ character will not be sent but the transmitter will be restarted.

The timer can be started by subprogram TX_ISRx and restarted by subprogram REAL_CLK. The timer is cleared by subprogram RX_ISRx when an host ACK is received or by subprogram REAL_CLK when the time out occurred with the restart transmitter option enabled.

In additon an event is sent to the host whenever the timer times out if the corresponding interrupt mask is enabled.

The handshake timer is also used to time the device X-ON/X-OFF event. If enabled and if the timer times out before the X-ON is received, an event will be sent to the host.

9.1 16-bit Second Timer

The 16-bit second timer is very similar to the 8-bit second timer. The timer is implemented by using 3 bytes, one for the 10 millisecond count down and 2 for the second count down. The major difference is that a 16-bit quantity is used to count down the second. This allow for a large time out for the no activity timer.

The no activity timer is used to disconnect the modem when no transmit or receive activity occurred. The timer is started by

subprogram REAL_CLK when the link is finally connected. The timer is reset by subprogram RSET_ACT whenever a receive or transmit interrupt occurred. The timer interrupt is processed by subprogram REAL_CLK.

EVENT PROCESSING & REQUEST MANAGEMENT — CHAPTER 10

The management of host requests and asynchronous card events is centered around the Event Manager, the RTS Queue, and the HP-CIO backplane protocol. Each RTS order from the host causes a decision by BCRTS to be made regarding what is the highest priority "thing" to be done. If the top of the RTS Queue is of higher priority than the current process, the current process is placed in the RTS Queue, and the top of the Queue is popped and passed to the host. Otherwise, the current process is reported to the host.

New requests, via the CLC order may be initiated at any time the card is SRQing for an order. Once initiated, the progress of the request is controlled by puting SWItch events on the RTS Queue. A request which is suspended is eventually made active by this process. Once the active request, a request may suspend itself by clearing the active task block (calling the routine "NOTHIN" (&BCRSR)).

Unsolicited events are passed to the host via the same RTS Queue mechanism, with the limitation that only one event per port can be on the RTS Queue at a time. Additional events are held on the Event Queue pending an Event Acknowlege to move them to the RTS Queue.

The Event Manager is called directly by anyone wishing to add any kind of event to a Queue to the host. It figures out what type of event it is, whether unsolicited events are disabled pending an AEK, and adds the event to the appropriate Queue. Passed to the Event Manager is the event, packaged in the form of an Event Block. This contains the event code, event priority, the port number (converted to a port ID by the Event Manager), and any additional information to be passed to the host. This block is pointed to by HL.

An interface routine is available to aid in the packaging of most events to the Event Manager. This routine, SET_EVNT, is passed the above parameters in registers B, C, D, and E, allocates a blank Event Block, fills it in, and calls the Event Manager. All events, with the exception of Message Received, Alert-1, etc. which have addtional information, use SET_EVNT.

This Queued scheme is used to reduce the time it takes to decide what to do next when an RTS comes along. The Queues are maintained in sorted order by priority. It is a simple matter to compare the

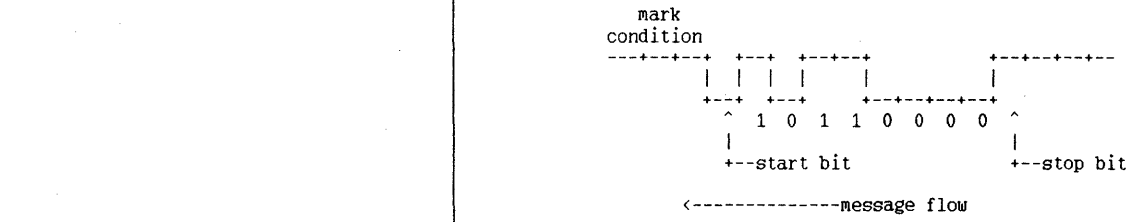
top of the Queue with the active priority and ship the host whichever is higher.

The scheme is complicated when an abnormal condition occurs. Whenever an event is cancelled (an abort is received for a task which has a SWItch in the RTS Queue, an unsolicited event in either Queue is cancelled when a read of the status register is made, etc.) the Queues must be searched and the event deleted before the host has a chance to read the event. (Hosts are easily confused.) For this purpose, a routine (EDIT_IT) is available to clean up the Event Queue and the RTS Queue for unsolicited events; status events are deleted directly in WTCABT (&BCWTC).

SPEED SENSING	CHAPTER 11

The speed sensing on the MUX is done by using firmware with some of the hardware features of the SIO. The SIO is programmed in the synchronous mode with a X1 clock running at 19200 bits per second and with 8 data bits for a character length. The SIO will start to generate interrupts to the firmware for processing as soon as it is synchronized with the incoming data. The firmware will start counting the number of bits for each level and the number of transition to determine the baud rate.

To really understand the speed sensing mode, the reader must understand how the serial data is sent to the SIO. In the diagram below, a data element always start with a start bit, follow by the data bits, and end by a stop bit. There may or may not be any parity bit.

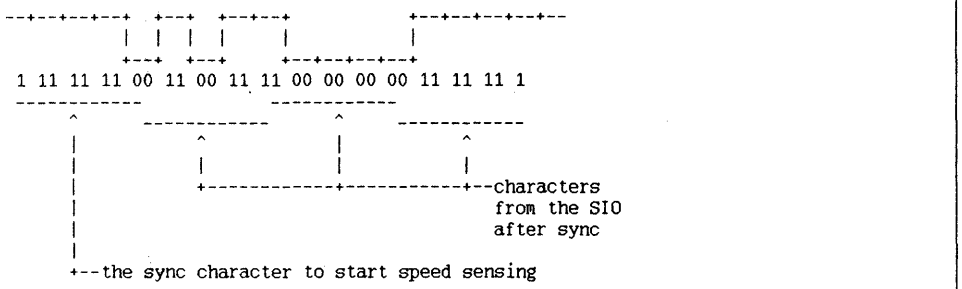


A CR character in 8 data bit mode with no parity.

The SIO is programmed to synchronize on the character 11111110B. By looking at the above diagram, one's are being generated until the start bit is received.

The MUX firmware uses the CR character for speed sensing. You will note from the above diagram there are five transitions in the level. The firmware will count the number of zero's in the first transition which is the start bit. The number of one's in the second transition, the number of zero's in the third transition, the number of one's in the fourth transition, and finally the number of zero's in the fifth transition.

For a CR coming in at 9600 baud the SIO will see the following characters after synchronization.

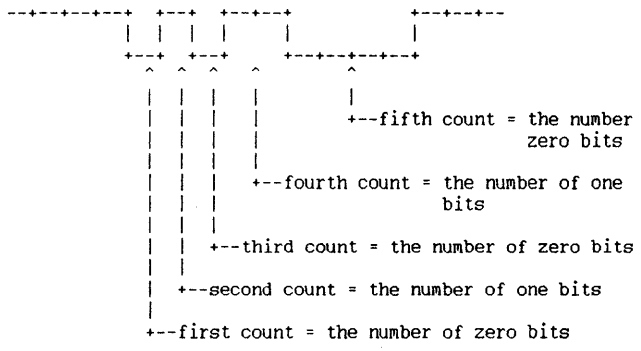


char#	char	
1	0110 0111	
2	1000 0000	
3	0111 1111	(the firmware will stop counting bits here after the last zero)

Similarly for a CR coming in at 4800 baud the SIO will receive the following characters after synchronization.

char#	char
1	0001 1110
2	0001 1111
3	1110 0000
4	0000 0000
5	0001 1111

The procedure can be continue for the other baud rates. The odd baud rate like 7200 baud does not have nice transition break and may not be detected on the first try. There is not guarantee.



incoming baud rate	first ct(0)	second ct (1)	third ct(0)	fourth ct (1)	fifth ct(0)
19200	0	1	1	2	3 or 4
9600	1	2	2	4	6 or 8
*7200	2	3 (2.67)	3	6	9 or 12
4800	3	4	4	8	12 or 16
*3600	4	5 (5.33)	5	10	15 or 20
2400	7	8	8	16	24 or 32
*1800	10	11 (10.67)	11	22	33 or 44
1200	15	16	16	32	48 or 64
*900	20	21 (21.33)	21	42	63 or 84
600	31	32	32	64	96 or 108
300	63	64	64	128	192 or 256
150	127	128	128	256	384 or 512
*134.5	142	143 (142.75)	143	286	429 or 572
*110	174	175 (174.55)	175	350	525 or 700
75	255	256	256	512	768 or 1024
50	383	384	384	768	1152 or 1538

Note that the baud rates tagged with an "*" are the odd baud rate which does not have nice sampling rate at 19200 bits per second. The sample rate will give the bit count specified in the parentheses under the second count. The count given is the one used by the firmware. The firmware is able to detect the baud rate some of the time depending on the source.

The 110 baud rate is another odd baud rate but the firmware is able to detect its present most of the time. It is able to do this

because the firmware does not check for an exact match on the counts.

To further insure a better detection rate a tolerance band is used to see if the incoming count matches a entry given in the table. For baud rates from 50 to 600 the tolerance is plus or minus 5. For baud rates from 900 to 1800 the tolerance is plus or minus 2. For baud rate 2400 to 9600 the tolerance is plus or minus 1. For 19200 the count must match exactly.

By experimention with a real terminal and with a real user, the firmware was able to detect the right baud rate. Also there is no false detection when any other characters are pressed beside the CR. However, the odd baud with the exception of 110 baud was not tested or received only limited attention.

The fifth count can have two different values as shown in the table. The first number is for a data byte which is 7 bit long with no parity or 7 bit long with even parity. The second number is for a data byte which is 8 bit long with no parity or 7 bit long with odd parity. The firmware will return the 8th bit as being an one if the fifth count matches the first number and as being a zero if the fifth count matches the second number.

An acid test was performed to determine how many ports can speed sense at the same time. The test is done by using one terminal sending input into all 8 ports. We found that at least 2 ports can successfully complete the speed sensing. The remaining ports will need the CR input again.

The speed sensing is enabled by using the control card request with subfunction 6. When this request is received by the firmware, the interrupt vector for the receive character and the special condition interrupt will be changed from its normal interrupt service routine (ISR) (RX_ISRx and SPC_ISRx, respectively) to the speed sensing ISR's (SS_ISRx and SSB_ISRx, respectively). In addition, the transmitter is disabled, and the capability to generate external/status interrupt is disabled.

The normal ISR's addresses are restored in the interrupt table after the speed sensing is done or when the user disable speed sensing by issuing the control card request with subfunction 7. The SIO and the CTC associated with the port in question are reprogrammed to its previous values.

The SIO is put into the synchronous "hunt mode" when the control card request to enable speed sensing is issued. The SIO is also put into the "hunt mode" again when a receive character does not match any of the baud rate or if the SIO encountered an error condition like data overrun. The data overrun condition usually occurs whenever the firmware is not fast enough to process every

receive character for every port.

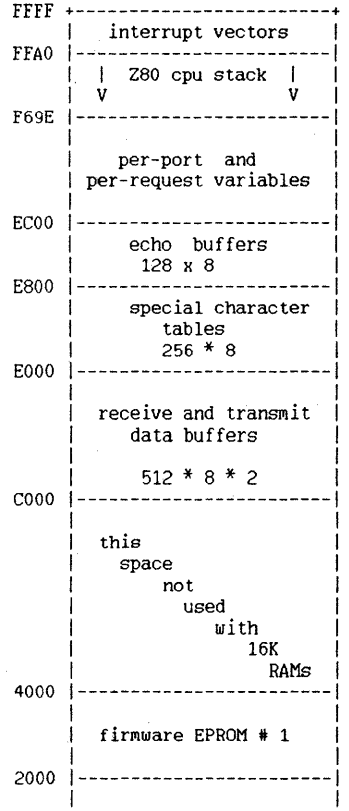
The following subprograms are used in speed sensing, directly and indirectly.

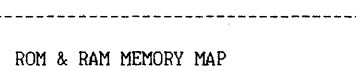
subprogram	source	description
RX_LOC	&MUXIV	A table containing the addresses to the interrupt trap cell for the receive ISR address. Index by the port number.
RX_VEC	&MUXIV	A table containing the normal receive and special condition ISR addresses for each port.
SS_VEC	&MUXIV	A table containing the speed sensing receive and special condition ISR addresses for each port.
CCD_BEG	&MXCCD	Subprogram to enable and disable speed sensing by using the control card request.
RESYNC	&MXCCD	Subprogram to program the CTC and SIO for the synchronous "hunt mode" used for speed sensing.
PRG_CTC	&MXWCC	Subprogram to program the CTC when the baud rate is changed or detected.
SET_SIO	&MXWCC	Subprogram to program the SIO when the character length, the number of stop bits, the parity, the card LED state, the hood LED state, the front-end drivers, and the internal loop back state are changed.
SPD_SEN	&SPDSN	Subprogram which have the speed sense algorithm.
SSB_ISRx	&SSBIR	Subprogram to reprogram the SIO into synchronous "hunt mode" after an error condition (usually data overrun).
SS_ISRx	&SSISR	Subprogram to set up the environment to call the speed sense subprogram after a character is received on the SIO.

The speed sense algorithm uses 16-bit counters to count the number of bits. If the character is not a CR, the SIO may or may not continue to receive all one's for each character continuously. The algorithm handle this problem by noting that the 16-bit counter should never be negative. If the number turns negative, then the algorithm treat this as a time-out and will cause the SIO to be reprogrammed into the synchronous "hunt mode" to search for the next sync character to start speed sensing again.

ROM & RAM MEMORY MAP — CHAPTER 12

The Mux card uses two 8K x 8 bit EPROMs and eight 16K x 1 bit dynamic RAM chips for memory. The Z80's memory address space is layed out as follows:



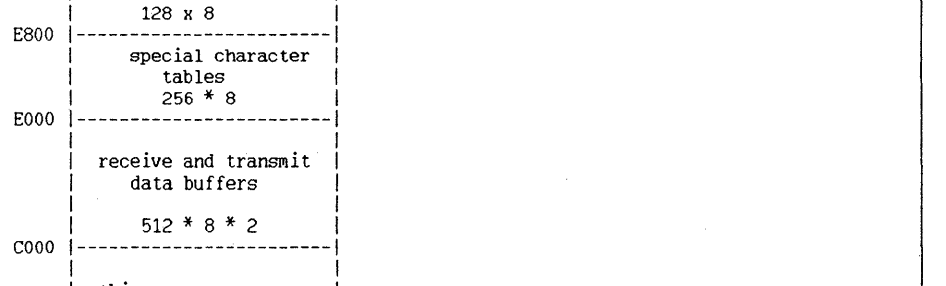


The entire Mux firmware resides in the two EPROMs. No code is downloaded to the card, nor does the resident firmware incorperate a downloader.

The character buffers begin at the first location of RAM, C000H. This is fixed so that the address of any character, buffer, etc can be computed as:

where: port # is the physical port # (0-7)
T/R is Transmit or Receive (1=T, 0=R)
index is the offset to the character in the buffer

Record headers are placed within the buffers to define record boundaries. For receive records there is a 3 byte header:



where: total length = the length of data + header; max 255.
E = this record terminated with error (parity, overrun)
tm code = termination code (ended on count, STT, etc)
term char = if STT termination, the character detected char

A non-terminated record has a zero length in the header. The one byte total length field, plus the three byte header length, gives a maximum data field size of 255-3 = 252 characters. This determines the card's data blocking factor.

The transmitter has a 1 byte header. This indicates the length of the DATA portion of the record (does not include the header). A

zero header indicates no record present.

The special character tables (one table of 256 bytes for each port) contain an attribute code for each of the possible character codes. When a character is received, it is ANDed with a mask to remove the parity bit, and used as an index into that port's special character table. The contents of that entry indicates what type of character it is. This entry is then ANDed with the toggle mask (TOGLMSK) to isolate those functions which are currently enabled (via WCC 1 or RDD subfunction). A zero result indicates that the character is simply a character (i.e. not anything which would require extra processing) and the character is placed in the user's data buffer. This speeds the receive ISR.

If the special character table entry is non-zero, and one or more of the same bits are set in the toggle mask, the further processing is needed. Note the bits defined align with the corresponding bits in the read configuration (WCC 1) and the RDD toggle subfunction.

Bit 7: This is a Handshake character (ENQ/ACK/XON/XOFF)
Bit 6: This is a Signal character (control-y, UNIX 'quit')
Bit 5: This is an Edit character (BS, DEL)
Bit 4: not used
Bit 3: This character is Quotable (BS, DEL, control-d)
Bit 2: not used
Bit 1: This is a Single Text Terminator
Bit 0: not used

Firmware Structure	CHAPTER 13

13.1 Overview

The firmware is divided into three major blocks. This was done to organize the operation of the Mux and to provide future products with a solid set of services to leverage from. (The most important is the Backplane to Midplane interface, described later.) The three layers are:

- o Backplane
- o Midplane
- o Frontplane

Each layer has a defined set of services which it provides, and defined interfaces to these services.

13.1.1 Backplane

This layer processes the orders and commands received via the BIC routines. It manages the card's Request and Event resources, and maintains the backplane protocol with the host. The Backplane layer makes any conversions necessary to interface host formats with those used on the rest of the card, for example, converting record lengths from high byte first to low byte first format. The PID (port ID) is converted to/from a port number also.

Major blocks within this layer are the order processors for WIC, RTS, WTC, BIC/MIC support, IDY, RD and WD, RSR, and the Event Manager.

13.1.1.1 BIC/MIC support

This layer contains the Interrupt Service Routines (ISRs) to which are vectored all interrupts from the BIC/MIC chip pair. This includes Order and Command interrupts from the BIC, and DMA termination interrupts from the MIC.

This is contained in module BPISR.

13.1.1.2 WIC (CLC) processor

Briefly, WIC (called via the Connect Logical Channel order) initiates the processing of new requests by reading in the Request Block, setting up the appropriate data structures, and calling the midplane. The status returned by the midplane tells WIC what state to leave the request in, which is communicated to the Event Manager.

This is contained in the module BCWIC.

13.1.1.3 Event Manager

The Event Manager is the module through which the midplane sends all events and request status changes to the host. Status changes go directly to the RTS Queue, while Asynchronous events go to the Event Queue pending an Asynchronous Event acKnowlege (AEK, via WTC).

This is contained in the module MUXEV.

13.1.1.4 RTS processor

The RTS order processor is the "back end" to the Event Manager. Whenever an event or status change of higher priority than the current task is at the top of the RTS Queue, the RTS processor de-links that event from the RTS Queue and passes it up to the host. Otherwise the current process is reported. The appropriate status tables are updated to reflect any changes to the current process.

This is contained in the module BCRTS.

13.1.1.5 RD and WD processor

The RD and WD order processors, actually one processor which serves both orders, initiates the actual data transfer from or to the port's frontplane data buffer. It then calls the midplane to direct what happens next within that transaction.

This is contained in the module BCRWD.

13.1.1.6 RSR processor

The RSR order processor transfers up to the host the current contents of the request's status block and terminates the request. The request's request block is de-allocated, and the state information regarding the current process is cleared.

This is contained in the module BCRSR.

13.1.1.7 WTC processor

The WTC order processor handles request aborts, event acknowleges, end of data status, and the Resume after an Error Trap. Aborts remove the indicated transaction, if it still exists. An Asynchronous Event acKnowlege moves an event from the Event Queue, if one exists there, or enables that port to place an event directly on the RTS Queue. End of Data calls the midplane to perform its end-of-record processing. This MUST be used while the indicated TID is the active transaction. Lastly, Resume is used after an ERror Trap to activate a paused transaction.

This is contained in the module BCWTC.

13.1.1.8 IDY processor

The IDY order processor returns the card's Identity block to the host. There is no effect on any other card process or function, so IDY can be requested at any time.

This is contained in the module BCIDY.

13.1.2 Midplane

The midplane processes the requests handed it by the backplane interface. Each type of request has a separate midplane processor, each with separate initiate, continue, abort, and end entry points.

All midplane modules maintain the RSR status block with the count of data transferred, record type, errors, etc.

13.1.2.1 Read Device Data

The Read Device Data midplane routine handles the reading of information received from the device via the frontplane. When given an initiate entry is processes any flushing of buffers or events, then sees if there is any data available to complete the request. If so it sets up the pointers in the request block for the RD processor, otherwise it signals a suspend to wait for data to arrive.

A continuation entry, caused by each part of a record being transferred to the host by the RD processor, sets up the pointers for the next transfer, suspends waiting for still more data, or signals the termination of the request by setting "end" status.

13.1.2.2 Write Device Data

The Write Device Data midplane routine handles the writing of information to the device via the frontplane. It is responsible for appending CR/LF to buffers, if so directed. On return to the backplane it indicates whether the transaction can continue or not depending on how much space is available in that port's transmit buffer.

Before indicating "continue" to the backplane, the pointers in the request block are set to the appropriate position in the port's transmit buffer so that the WD processor will operate correctly.

13.1.2.3 Control Device

The Mux does not support any Control Device requests. Any access to this module returns Error Trap status, with the RSR status block indicating an illegal request.

13.1.2.4 Read Card Information

The Read Card Information midplane routine handles the reading of configuration information contained within the card. If the information read was the interrupt status, then the status gets cleared, and a routine in the backplane is called to remove the status blocks from the Event and RTS Queues.

13.1.2.5 Write Card Configuration

The Write Card Configuration midplane routine handles the writing of configuration data to the card. The routine also performs the configuration of the card, including supplying default values for some requests. All configuration is subject to some level of verfication check (length of data, range checking, etc.).

13.1.2.6 Control Card

The Control Card midplane routine implements those requests which are modes or one time actions, for example, enter speed sense mode, or flush receive buffers. The action is specified in the request subfunction field, as there is no data block on control requests. The action is executed immediately upon the initiate entry to the routine.

13.1.3 Frontplane

The Frontplane consists of the interrupt handlers for the various interrupts generated by the SIO chips. These are the External Status, Special Condition, Character Received, and Character Transmitted. Further interrupt service routines are included for the speed sense mode.

13.1.3.1 External Status

The External Status interrupt service routine handles the BREAK interrupt from the SIO. Since there are no modem line connections to this card, none of the other interrupts can (should) occur. If enabled, the occurrence of the BREAK interrupt causes the event manager to be called to send a BREAK message to the host.

This is contained in the module EXISR.

13.1.3.2 Character Received

The received character interrupt service routine gets entered for each character received by the card. The character is read, parity is masked, and the character is acted on according to its value. Non-special characters are placed in the port's receive buffer. Special characters (edit, termination, quote) trigger whatever action is indicated.

If a termination condition is detected, either by special character or by count (end on count, or buffer full) the current record is packed up (its header is filled in) and the record is made known to the backplane.

For speed, part of the Receive Character ISR is duplicated for each port.

This is contained in the module RXISR.

13.1.3.3 Special Condition

The Special Condition interrupt service routine gets entered whenever a character is received which is somehow in error. This may be because of bad parity, a framing error, or from SIO fifo overrun. Depending on the options set, either the character is ignored, placed in the buffer, or replaced by another character and placed in the buffer, and the buffer is either left open, or packed up and sent to the backplane. The function of placing the character in the buffer as is is performed by jumping into the

normal received character ISR for that port.

This is contained in the module SPISR.

13.1.3.4 Speed Sensing

The speed sensing function of the frontplane is divided into three modules, SPDSN and SSISR, and SSBIR. SSISR is an interrupt service routine which is entered whenever a character is received while that port is in speed sense mode.

This is accomplished by moving the interrupt vector for that port to point here instead of the Received character ISR. The ISR then calls the main routine which processes the character. Briefly, the speed sense function works by programming the SIO for that port to synchronous mode and thus sampling the line 8 times per byte received. The length of each sequence of ones and zeros is timed (i.e. counted) and compared to the known sequences for the Carriage Return character at different BAUD rates.

SSBIR contains the re-synchronization code for handling SIO overruns during the speed sense mode.

13.1.3.5 Transmit Character

The Transmit character interrupt service routine sends a port's data to the SIO on an interrupt basis. Each time an interrupt is received, the next character is transmitted, either from the port's main data buffer, or from the Echo Buffer. The ENQ/ACK counter is maintained, and when it rolls over a handshake is performed, if enabled.

When all data in a data buffer has been transmitted, an event is generated (if enabled) to the host, or a suspended write is awakened, by calling the Event Manager.

13.2 The Backplane

The following are detailed descriptions of the code within each backplane module. They are intended to be read together with the source listing, and to be used as a supplement to the comments contained within the source.

13.2.1 Identify: BCIDY

The purpose of the IDY routine is to return the IDY block to the host.

o The MIC DMA can only do transfers to/from RAM. Therefore the IDY block must be moved to RAM before being transferred.

13.2.2 Event Manager: MUXEV

The Event Manager module contains several routines, as follows:

13.2.2.1 FREE_EVB

This routine puts an event block (pointed to by HL) back on the list of free event blocks. While playing with the pointers to the freelist, the interrupt system is turned off to prevent trouble. When done, the interrupt system is restored to its former state, on or off.

13.2.2.2 GET_EVB

This routine allocates an event block from the list of free event blocks. If there are none available a hard jump is taken to DIE, since there is nothing we can do about it. This should never happen since there are lots of event blocks lying around.

The address of the allocated block is returned in HL.

13.2.2.3 EVNT_MGR

This is the core of the event manager. EVNT_MGR takes an event block (pointed to by HL) which has been filled out by someone else, and places it on the appropriate queue. If the event is a status change, it is placed on the RTS Queue. If it is an event, and events are enabled on that port, then it is placed on the RTS Queue and events are set disabled. Otherwise, events are placed on the Event Queue, pending a friendly AEK (via WTC) to move one of them to the RTS Queue.

o Events to the host are given to EVNT_MGR with the port number in the PID field. Since the host expects a port ID, not number, EVNT_2 changes the port# to a PID.

o ADD_Q puts an event block on either the RTS Queue or the Event Queue, in sorted order by priority. Both queues must be sorted to allow events to have priority over status changes, and to emulate the ASI card's priority relationships between different card events (Break is higher than Signal, for example).

13.2.2.4 SET_EVNT

This routine is a convenient way to call EVNT_MGR. It allocates an event block and fills it in with the passed parameters. The only restriction is that events with more than just an event code and port# cannot use this routine (no more registers to pass parameters).

o On a status call, DE points to the request block, the first two bytes of which contain the TID of that request. This is LDIR'd to the event block.

13.2.2.5 EDIT_Q

One of the problems with a queued backplane is that when something happens which changes the state of the world (e.g. a RDD happens just after a message received interrupt was posted), there may be information which is on its way, via one of the queues, to the host which is no longer valid. This information must be removed from the queues to avoid confusing the host. EDIT_Q searches the queue pointed to by DE to find an event (not a status) which matches that which was passed to it. If it finds one, the links to it are moved around it, and the block is returned to the freelist.

EDIT_Q is not external since it is only to be called from EDIT_IT.

13.2.2.6 EDIT_IT

EDIT_IT is an interface to EDIT_Q to delete events from the Event and RTS Queues. First it sets the port id (not port number, since EVNT_MGR changed that to PID before sticking it on the queue), then it tries to edit the RTS Queue.

o If something was deleted from the RTS Queue we have to see if there is anything in the Event queue which should be moved to the RTS Queue. The way this is done is to simulate an AEK. This isn't done if there are no more events waiting on that port, to save time. The event queued flag is reset so that WTC_EAK doesn't get confused.

o If nothing was deleted, the Event Queue is tried.

13.2.3 Read Transparent Status: BIC_RTS

13.2.3.1 BIC_RTS

This module processes the RTS order. Here the host is asking "what do you want to do next?". There are several possiblilities. If no task is active, and there are no events pending, then the Idle response is given. If the current process has a higher priority than the event block at the top of the RTS Queue, then the current status is reported. Otherwise, the current process is moved to the RTS queue, and the top of the queue is removed and sent to the host. If the top of the queue implies a switch to that process (either a switch or an end of data), then that process is made the active process. Note currently all processes (requests) have equal priority, and all events have priority over all processes. This need not be the case; if there is some reason to change the current values, any process or event can have any priority.

o BIC_RTS Checks to see if there is an active transaction. If so, the active priority is compared to the priority of the event at the top of the RTSQ. If the Queue priority is higher (lower number) then an the current transaction is moved to the RTSQ by allocating an event block and copying the current transaction's status block to the event block. This means that the active status block must be in the form of an event block.

13.2.3.2 UPDTID

This routine sets the state of the active transaction to match the RTS block being sent to the host. If the RTS block is a SWItch, then the TID to which the switch is being taken becomes the active transaction. Similarly, if the RTS block is an end of data, a switch is implied, and the TID becomes the active TID. The end of data implies a switch because of the implementation of the backplane protocol on the HP-9000 machines.

If the RTS block contains an asynchronous event, then the bit in that port's event status corresponding to that event must be cleared. This is done by indexing into a table of masks based on the event code. Since, by our conventions, an event code has a one to one correspondence to the bit in the interrupt mask which enabled it, the indexing can be used. Otherwise we'd have a mess. Note that the event block contains a port ID, not port number, so a conversion must be made in order to find the correct port's status.

If the RTS block does not result in a switch to a transaction (i.e. is not a SWItch or End Of Data), then the active transaction is set

to nil. This is because the act of asking for transparent status by the host (the RTS order) takes the card out of whatever transaction it was in, and back to "subchannel" mode. Another RTS will follow any RTS reporting an event, and on Vision machines you will get an RTS following End of Data, also.

13.2.3.3 ORD_EXIT

This is a place where all (or at least most) order processing routines exit. It initiates an SRQ for the next order and then returns from the BIC interrupt.

13.2.4 Write Transparent Control: BIC_WTC

The Write Transparent Control module processes the WTC order, and its subfunctions AEK, RES, EOD, and DLC.

13.2.4.1 BIC_WTC

This is the entry point from the BIC interrupt processor. The WTC block is read into a dedicated buffer in RAM (there can be only one WTC in process at any time, so this doesn't have to be per port). From there the subfunction is fetched and decoded. The appropriate routine is then executed.

13.2.4.2 WTC_RES

This routine implements the transaction resume function. The TID is searched for, and if not found the entire WTC operation is considered a NOP. It is assumed that the transaction is not currently active (that an Error Trap had previously occurred), so when a continue event is generated and placed on the RTS Queue that it is the ONLY status event for that transaction.

13.2.4.3 WTC_END

This routine implements the End of Data function of WTC. Since the only midplane process to use End of Data is WDD, we assume that that is the midplane process intended. After finding the indicated TID's request block (and therefore his port's stuff) we can call the midplane. The HP-9000 does not use this feature of the card, but the Vision folks might.

13.2.4.4 WTC_EAK

This implements the Asynchronous Event acKnowlege function of WTC. When an AEK is received it means that an event had previously been sent up to the host via RTS, and that the host is signifying that it is able to receive another event. What we do then is to search the Event Queue for an event posted on the port which has just been acknowleged. If one is found it is moved to the RTS Queue. Otherwise the Events Enabled flag is set signifying that if an event occurs it may be placed directly on the RTS Queue.

o The exception to the above is if the EV_QUED flag is set, then we already have an event on the RTS Queue. This can happen if the host gives us any extra AEK's (which according to the protocol standard should be ignored).

13.2.4.5 WTC_ABRT

This is used to abort a pending request. The major task here is to search the RTS Queue and delete any (the one) reference to this request, if it exists.

An status event is generated to the host informing it that the request is gone. Note that the standard defines an abort situation as a request, not a demand. We could choose to ignore the abort, if we wanted to. However, the HP-9000 implementation of the protocol can't handle a delayed abort, so we force generate the event here.

The midplane is then called to clean up anything it may have going. Note that it cannot make references to its request block, since that is now gone.

Lastly, the request block is de-allocated, and if the request happened to be the active request, the active status block is set to idle.

13.2.4.6 FIND_TID

This subroutine searches for the TID specified in bytes 1,2 (counting from 0) in the WTC buffer. The address of that request's request block is returned in DE. The Z flag is set if no block is found, so a JP Z will branch if the request doesn't exist.

13.2.4.7 GET_STUF

This subroutine takes a port's ID (not port number) in A and computes the address of the "PORTSTAT+1" entry in his port's stuff. This is returned in HL.

13.2.5 Connect Logical Channel: BCWIC

This module processes the Connect Logical Channel order (which used to be called Write In-channel Control, WIC).

13.2.5.1 BIC_WIC

Here the CLC data block is read in to an allocated request block. The port ID is converted to a port number so that the midplane knows who to talk to.

There are three pointers in each port's stuff which point to the request block for read device data, write device data, and all other requests. This allows one of each type of request to be pending at the same time. Each port's stuff points to the request block(s), and each request block has a pointer to that port's stuff. This makes it easier to find one, given the other.

13.2.5.2 WIC3

Here the request's data length parameter is converted to midplane format (low byte first). A check is made for illegal requests, and if ok, the midplane is called.

13.2.5.3 MIDP_RET

Here the midplane has returned with a condition code indicating what to do with this request. If the Carry bit is set the midplane is indicating that the transaction can continue whenever the host gets around to doing an RTS, and when all higher priority housekeeping is done. The Z flag is set if the request is blocked for some reason, for example, no buffer space for a write request. Otherwise it is assumed that the midplane is indicating that an error exists in one of the parameters of the request, and that an ERT should be generated. The order of checks used here is assumed by some of the midplane routines, in that they may not clear all unused flags. (For example, both the C and Z flags could be set. The midplane's assumption results in a "continue" interpretation.)

In the case of Continue (now called SWItch) a "SWI" status event is generated, and the event manager is called. ERT results in an Error Trap status event being sent to the event manager. A SWI status is not generated for ERT or Idle (the Z flag) thus pausing the request.

13.2.5.4 ERR_005

The check here is for WCC 34, which sets the port ID. Since the port ID is probably not valid for this request, we can end up here. Putting the check for WCC 34 here makes the straight line path earlier a bit faster (the usual request is not a WCC 34). Since the Midplane doesn't access the port's stuff for this request we can allow a WCC 34 which happens to use an already assigned port ID to use the straight line code without trouble.

13.2.5.5 FREE_RQB

This subroutine is used to return a request block (pointed to by HL) to the free list. Make sure that the request has been aborted or otherwise removed from the card before calling this routine.

13.2.5.6 GET_RQB

This routine fetches a request block from the freelist. The entire block is zero'd out, so the default RSR status gets cleared. The block's address is returned in HL.

13.2.6 Read Request Status: BCRSR

This routine processes the RS order.

o It is assumed that the host has set the "disconnect" bit in the RS order to remove the transaction.

13.2.6.1 ABRT_REQ

This subroutine is called by RSR and WTC ABT to remove a pending transaction. The pointer to the request block in the port's stuff is cleared (so WIC won't think it's busy), and the pointer in the request block to the port's stuff is cleared (so FIND_TID won't think its in use).

The request block is then returned to the freelist.

If the request being aborted happens to be the active transaction (this will always be the case for RSR) the active status is also cleared.

13.2.6.2 NOTHIN

This subroutine cleares the active status block.

If you are the active transaction and want to suspend, just call this routine.

13.2.7 Read Data and Write Data: BCRWD

This routine handles both the RD and WD orders.

Based on the request code, the appropriate I/O routine is called.

o A zero length transfer cannot be handled by the BIC/MIC I/O routines. This routine fakes zero length transfers as 1 byte; the RSR block will have the correct length of zero.

The midplane is then called at its Continue entry point.

13.2.8 BIC & MIC Interrupt Service Routines: BPISR

The BIC interrupt service routine services all the interrupt generated by the BIC. The interrupt from the BIC is actually generated by the MIC directly. The BIC in this case will be the device with the lowest priority in the interrupt chain. This will ensure that the SIO receivers and transmitters interrupt will be processed in a timely manner where possible.

The BIC and MIC ISR will run with the interrupt enabled at all time when possible. The registers are saved by pushing the contents onto the stack.

The host must perform a subchannel connection before performing any transaction. The MUX firmware will not verify that the subchannel is already connected. If the subchannel is not connected, the BIC will not be able to generate any SRQ to the host even if the firmware tells the BIC to do so. The reason for this is that the SRQ address present bit in the BIC register 1 is not set.

Only one interrupt condition per interrupt is processed. The other interrupt conditions will cause another interrupt to the ISR after a return from interrupt instruction is executed.

The IFC or DCL signals will immediately reset the card and causes the firmware to began execution at address 0. The MUX firmware does not look at these signals. However, the self-test firmware will check these signals to determine whether self-test should be perform.

Upon receiving a BIC interrupt, the service routine will read the interrupt latch from the BIC register 5 and the interrupt mask from the BIC register 6. The two values are masked together to determine which interrupt should be process.

The status register empty (SRE) interrupt has the highest priority in the processing order. This interrupt is enabled if a status code is put into the ARQ_BUF queue when the host has not read the last one.

The MUX card will never generate a nonmaskable interrupt to the host. Therefore, the card should never receive the nonmakable interupt acknowledge (NMK) interrupt from the host.

The request attention (RQA) interrupt is sent to the host when the card is unabled to process the command fast enough. The card will send the ready for command (RFC) status to the host to request the next command. This interrupt is always enabled right after reading a command. A race condition could exist where the host sends a command right after the RQA bit in BIC register 5 is checked.

The end condition (END) should only occurred during the data transfer mode and will be process by the MIC DMA ISR.

The FIFO read condition (FFR) occurs only if the data overrun on the host write. If this case the BIC/MIC ISR will read and discard the data byte until the end condition.

The order and command interrupt conditions are processed by using a jump table to go directly to the correct processing subprogram. The MUX firmware processes the order interrupt condition first before the command. The reason for this is to save some processing time because orders will be received more frequently then commands.

Each order and command have a separate processing routines. Some of the subprograms are external to the BIC ISR and some of the shorter subprograms are part of the ISR. The external subprograms are described above.

If undefined orders or commands are received or if a illegal request for starting a transaction is received, the protocol error status code will be return to the host by the subprogram BIC_ERR. At this point the host should reset the card before continuing, although in most cases this may not be necessary. No attempt was made to find out when this is not necessary.

The protocol error and the dead-or-dying error status code will cause the card not to do a SRQ for the next order. The ISR will only do a return from interrupt. This is done to allow the idle loop to run to get information from the RAM to see why the card failed.

The MUX firmware will only process the abort (ABT), the subchannel connect (SC), and the resume (RES) commands.

The MUX firmware will only process the identity (IDY), the pause)PSE), the subchannel disconnect (DIS), the read request status (RSR), the read (RD), the write (WD), the read transparent status (RTS), the write transparent control (WTC), and the write inchannel control (WIC) orders.

13.2.8.1 BIC_ISR

This routine controls the interface to the HP-CIO BIC gate array. The exchange over the HP-CIO backplane is defined in the BACKPLANE INTERFACE CIRCUIT (BIC) by Bill Martin, the BIC PROGRAMMERS REFERENCE MANUAL by Bill Martin, and the HP-CIO STANDARD BACKPLANE PROTOCOL FOR SMART CARDS by Greg Dolkas.

13.2.8.2 BIC_EXIT

This is the main exit point from the BIC/MIC ISR. The registers are restored to its original content on entry and a return from interrupt is performed.

The DI and EI instructions were necessary due to a MIC bug of not recognizing the second RETI if the previous instruction was an RETI.

13.2.8.3 ARQ_HOST

This subprogram sends the ARQ status code to the host through BIC register 2. If the BIC register is busy, queue the status until the host is ready for it. Subprogram SRE_RTN will send the next status code when the register is ready.

13.2.8.4 BIC_ABT

This subprogram process the abort command by acknowledging it, clearing the necessary flags, and clearing the SRQ address register in the BIC (register 1).

13.2.8.5 BIC_ERR

Send the protocol error status code to the host and just do a return from interrupt. This is only sent when a backplane protocol error is encountered.

13.2.8.6 DIE

Send the dead-or-dying status code to the host when the MUX firmware encounter a unrecoverable internal error. This should never happen, if it does we got big problem.

13.2.8.7 BIC_DIS

This subprogram process the disconnect order by clearing the SRQ address register in the BIC.

13.2.8.8 BIC_INIT

This subprogram initialize the BIC and MIC on initial start up. The subprogram will also spin in a loop until the perpherial address bit is set indicating that the card has been sense by the host. The ARQ buffer pointers are also set before returning.

13.2.8.9 BIC_PSE

This is the PAUSE order processor. The normal condition is to set the pause bit and exit the ISR. But due to a race condition, it will first test if there is anything on the RTS queue. This can happen if the pause order is received while interrupts are disabled in the event manager (ADD_RTS, to be specific). If so, the test for "are we paused" will fail, and the card will get stuck with a "continue" event on the RTS queue, and nor SRQ (card paused). By testing the RTS queue here we prevent this situation.

13.2.8.10 BIC_RES

The resume command processor. If the card is in the pause state, an SRQ for the next order will be sent to the host.

13.2.8.11 BIC_SC

The subchannel connect command processor. If the subchannel is already connected, return a protocol error status code. Otherwise, get the SRQ address from the command and write it to the BIC SRQ address register (1). Then send the SRQ to the host for the next order.

Next enable the RQA interrupt in the interrupt mask. The reason for this is that a command can come in right after reading the interrupt status register. If the RQA bit is set, send the ready for command (RFC) status code to the host.

13.2.8.12 BIC_END

This routine is part of the data transfer subprogram. It will first send the MIC command to disable the DMA in case it is still active. At this point an active DMA will not transfer anymore data since the host has sent an END condition to the card. Next determine whether a host read or a host write is being perform.

If this is a host read and if the DMA was still active when the end condition is received, then a host read data underrun occurred. The host has terminated the transfer early. Set the S flag, clear the Z and C flag, and set up the condition to return to the order service routine requesting the data transfer.

Similarly, if the host read is active and if the FIFO ready bit in the BIC interrupt status is set, then a host read data underrun occurred. The processing is the same as described above.

For the host read if neither of the above condition exist then a normal completion is performed by setting the Z flag and clearing the C and S flags before returning to the calling programs.

The host write can have 3 conditions if a END condition is received. They are data underrun, data overrun, and bad BIC. There is a bad BIC condition if the FIFO ready bit in the interrupt status is also set. The condition requires the firmware to read the last byte from the BIC before continuing the processing.

The host write underrun occurs if the DMA is still enabled. In this case perform the same function as for the host read underrun.

If it is not bad BIC or underrun, then a data overrun condition occurred. Set the C flag and clear the Z and S flags, then set up the condition to return to the caller.

The normal completion of host write DMA is done through the DMA ISR.

13.2.8.13 BIC_FFR

The FIFO ready condtion without the END condition can only occur for a host write. This is a data overrun. Read the byte and discard until the END condition occur.

13.2.8.14 DMAB_ISR

This subprogram handles all the MIC channel B DMA interrupts. The subprogram save all the registers by pushing them onto the stack. The interrupt system is reenabled to allow the frontplane to continue processing interrupts when they occurred.

The exit point from this ISR is through BIC_EXIT.

The DMA ISR uses the MIC register 0 auto enable bit to determine whether a break occurred in transferring data from the circular buffers. If the bit is cleared, then a break in the circular bufferring have occurred. Set up the MIC DMA to transfer the remaining data.

If the BIC end condition is not set and if the auto end enable bit is set in the MIC, then a host write data overrun condition occurred. Clear the BIC FIFO and enable the BIC for FFR interrupts to throw data away until an END condition is encountered.

If the BIC END conditon is set then a normal DMA completed on the host write. Normally, the firmware will reinitialize the environment for the next DMA data transfer. But due to a bad BIC problem, code was added to check the BIC FIFO bit. If it is ready, then there is one more byte in the FIFO that has to be read.

13.2.8.15 HCIR_IO

All circular buffer data is transfer to or from the host by using this subprogram. This subprogram determines if the data transfer will wraparound on the circular buffer. If it does, turn off the auto enable bit in MIC register number 0 and compute the beginning address and the data transfer length.

This subprogram will call either HRD_IO or HWD_IO to start the data transfer. HRD_IO is called if bit 9 of the buffer address is cleared, and HWD_IO is called if bit 9 of the buffer address is set.

13.2.8.16 HRD_IO

Set up the BIC and flags for the host read data transfer. Program the MIC and start the data transfer.

13.2.8.17 HWD_IO

Set up the BIC and flags for the host write data transfer. Then program the MIC and start the data tranfer.

13.2.8.18 SRE_RTN

When this program is called, send the next ARQ status code to the host through BIC register 2. The update the buffer pointer and disable the SRE interrupt if there are no more ARQ status in the queue.

13.2.8.19 SRQ_HOST

This subprogram sends the SRQ to the host for the next order. There are two types of SRQ's. The first is sent by writing 10H to BIC register 4. This one is used when no data transfer occur. The second method of sending SRQ is by writing a 10H to BIC register 5 to clear the END conditon which was set because of a data transfer.

13.3 The Midplane

The following are descriptions of the modules comprising the Midplane.

13.3.1 MUX_CCD (&MXCCD)

This subprogram handles all the control card request to the card. The request is never suspended.

CCD_BEG is the main entry to start processing of the control card request.

The subfunction code to enable speed sensing will cause the card to generate a solicited interrupt or event when the baud rate is detected.

The transmitter will be restarted when the transmitter is stopped due to an device X-OFF or to the host waiting for ah host ACK.

CCD_ABT entry does nothing special.

13.3.2 MUXCDV (&MXCDV)

This subprogram does no special processing. It is included to satisfy the external entries for BIC_WIC.

13.3.3 MUX_RCI (&MXRCI)

This subprogram process all the read card information request. This subprogram consists of 4 entries -- RCI_BEG, RCI_CONT, and RCIABT.

RCI_BEG sets up the buffer address, the buffer length, and the read status for the requested information.

RCI_CONT does nothing except to set the active status to RTS_END to

terminate the transaction.

Note that all read card information request should not have the block bit set in the request block. The information transfer to the host cannot be blocked. The reason for not doing this is to save code space. The only request which will exceed the block size is the read card RAM request (subfunction 250).

RCI_ABT does nothing except to set the active sttus to RTS_ABT.

13.3.4 Read Device Data: MXRDD

This module implements the Read Device Data request.

13.3.4.1 RDD_BEG

This is the initiate entry point for RDD, which is called by BCWIC because of a CLC order with the request field set to 1.

On entry, BC points to the port's stuff, and IY points to the request block. Since this isn't convenient for some processing here, the addressing is switched; IY points to the port's stuff, and BC points to the request block.

o The first thing checked is whether or not a read is legal under the current receive configuration. This prevents someone from hanging a read on the port when the receiver is disabled.

o The next thing to check is whether a receive interrupt is pending on the host. This can happen if a read request is initiated on the card at the same time a termination condition is seen on the port's frontplane. The host's request, if it gets to the card before the event is reported to the host, will read the data, thus clearing the condition which caused the interrupt. Since the interrupt is queued in either the Event Queue or RTS Queue, these queues must be searched to remove the event. Note that this can "impact" performance if it is done too often.

o The request block is checked to see if the F bit is set. This directs the card to flush any data the port may have accumulated prior to the request. This is used by systems (e.g. the 3000) which don't listen to data comming in if a read request is not pending (except for signal characters). If a separate flush buffers request were performed there would be a window between the flush and the read where data could be entered under different configuration. (Remember the read subfunction can toggle some frontplane configurations.) If a flush is performed, then there is now lots of space left in the receive buffer, so we have to send an

XON if we had previously sent an XOFF.

o RDD_BG3 Now that all the housekeeping is done, on to serious business... The receive backplane pointer is checked (only need to check the high byte since it can't be zero if there is a buffer) for a buffer which has at least one byte in it. A previous read may have removed part of the data, but we don't care since the data offset pointer (RDBUFN) will have been left updated. If there is some data, then it, or part of it, will be used to satisfy at least the first block of this request. The read will not be suspended on this block.

o If there is no data, there are still two possible ways to not have to suspend this read. First is to check for alert-1 mode. If enabled, the read will not be suspended, even if we have to generate a zero length record to satisfy it.

o If there is no terminated data, and we're not in Alert-1 mode, then we see if there is enough data to satisfy the request sitting on the frontplane. If so, we terminate it now. The equivalent is if the read happened first, the data would have been terminated by Host Buffer Full. Since we shouldn't care which happens first, we have to make this check. From here there are two paths; either there is enough data or not. In both cases we will be changing the frontplane counters, and so have to call SET_CNTR. Since the frontplane downcounter has not expired, the counters which SET_CNTR uses have not been updated, we have to call UPD_EOC to update the end on count and host buffer full counters before calling SET_CNTR. Otherwise all sorts of nasty things will happen. So to save code, first we call UPD_EOC, then see if there is enough data.

o Note that we only need to do 8 bit arithmetic since the length of the buffer can't be more than 252 bytes long. If more data were entered, the internal counter would have popped, and the frontplane would have had a record already ready to go, and we wouldn't be here.

o If there is not enough data, the down counter for the host buffer size is updated to reflect how much more we need to satisfy the request.

o Now, if we ended up deciding that what was on the frontplane, if any, was enough to satisfy the request, we call the frontplane routine PACKITUP to terminate the current frontplane record. If, since the last read, the frontplane parameters were changed, the new values can now be installed on the frontplane. The initial values of the end on count down counter is set, and the host buffer size down counter is set large enough not to get in the way. Now, finally, we can call SET_CNTR and let the frontplane go.

o If we have to suspend the read, the host buffer size down counter

is set and SET_CNTR is called. Note the call is made AFTER the suspend flag is set, since SET_CNTR checks the suspend flag to see if it has to worry about the host down counter.

o The last thing we do before leaving in a suspended state is to toggle the frontplane functions based on the subfunction code. This is done by changing the toggle mask to enable or disable the functions which are allowed to be "special" in the receiver ISR. We suspend by setting the Z flag back to WIC.

o RDD_9 If a buffer was ready when the read was posted, or was made ready by the horseing around we just did, we end up here. Since we will be dealing with the request block for a while now, the addressing is switched back; IY points to the request block and BC points to the port's stuff. The interrupt system can come on since we are done messing aroung with the frontplane.

o RDD_14 First we set up the beginning address for the RD and WD processor (BCRWD). This is computed from the backplane receive buffer pointer plus the offset counter. The counter is initialized to 3 to account for the header bytes in the receive buffer. Note that we have to make sure that the math accounts for the 512 byte circular nature of the buffer by resetting bit 9, just incase it got set by a wrap.

o The length is computed from the total length of the buffer, minus what has already been transferred. "What has been transferred" includes the three header bytes. The actual length of transfer is the minimum of this value and the length of the user's buffer, which may be less.

The carry flag is set to tell the backplane that the request is ready to go.

13.3.4.2 RDD_CONT

This is the contination entry point for RDD, called by BCRWD after completing a transfer to the host.

o On entry, BC points to the port's stuff, IY points to the request block.

o The first order of business is to update the RSR block to reflect what has just taken place. The length parameter in the request block which RWD used for the transfer is used to update the transfer length field of the RSR block, which is at the end of the request block. The terminating code and characters are also copied to the RSR block. This leaves the type of the last block as the type of the entire transfer, which assumes that all blocks up to the last one were "partial records".

o The residual count is set based on what is left in the current backplane record. Again, after the last block this will reflect the correct overall value.

o There are now two possibilities. Either the last block sent to the host completely emptied the current backplane record, or there is still some data left in it. If there is no data left, this phase of the transfer is complete If so, then the host wants the rest of the data saved. We move the backplane offset pointer to account for what was transferred, then we're done.

o If there is no data left, or if the S bit is not set, we de-allocate the data buffer. Again we have two possibilities; either there is another buffer after this one, or there isn't.

o If there is another buffer, we check if the read is a blocked read. If so, and if the record is not a partial buffer (signifying that the read has completed) then we go set up to transfer the next block. A check is also made in case the data buffer was not a full 252 bytes. This can happen if a previous record was partially read, then another request, with a large buffer and the Block bit set, comes along. The first block would be short, so to agree with the Backplane Protocol Standard, we simulate a termination condition to force the request to complete after that short block.

o If the block just sent terminated the read, we have to generate an event to the host to inform him of the "new" record. This also sets data available in the status word, and other good things.

o RDD_5 If there isn't another completed buffer we zero out the backplane pointer to indicate no data available. Alert-1 mode is re-enabled on the frontplane if it should be (the frontplane's flag gets reset after the first character so that it does not get bogged down). If there is any data on the frontplane an alert-1 event is generated instead.

An XON character is also sent if we now have enough space and had previously sent an XOFF.

13.3.5 MUX_WCC (&MXWCC)

There are 3 major entries into this subprogram. They are WCC_BEG, WCC_CONT, and WCC_ABT.

All write card configuration requests are started by calling WCC_BEG. This subprogram will verify the subfunction code and the data transfer length. If the information are valid, the subprogram will set up the staging buffer address and the data transfer length. Note that the block mode bit in the request block should

not be set. None of the subfunction has parameter length close to the block size.

After the data is transferred from the host to the card, WCC_CONT is called to complete the transaction. The data content are verified whereever possible before being move from the staging buffer to the port stuff area for the port. If the data is invalid, then an error is returned to the host in the read status block.

The WCC_ABT entry does nothing except to set the active status to RTS_ABT.

13.3.6 MUX_WDD (&MXWDD)

This subprogram has 4 major entries. They are WDD_BEG, WDD_CONT, WDD_ABT, and WDD_END.

All write device data requests are started by calling WDD_BEG. WDD_BEG will check to see if there is enough buffer space for the transaction. If there is enough space, set up the buffer address and the data transfer length to continue the transaction.

If there is not enough space, suspend the transaction. The transmitter interrupt service routine will restart the transaction when enough space becomes available.

After the data transfer has completed, WDD_CONT is called to continue or complete the transaction. The subprogram will first update the transmission log. If the remaining data transfer length is not zero, go set up the buffer address and the data transfer length for the next write if there is enough space. Otherwise, suspend the transaction.

If the remaining count becomes zero, check to see if the output separator appendage option is enabled. If enabled add the output separators to the transmit buffer at this time. Now go clear the header of the next record and then set the byte count in the current record to make the record available for the frontplane for processing. Turn off the interrupt system and then call the frontplane subprogram to start the transmitter if it is not busy.

The abort write device data transaction request will cause the clean up of any suspended write device data transaction. Entry WDD_ABT will be called to do this processing.

WDD_END is called to terminate the write device data transaction early. The firmware will check to see if the output separator appendage option is enabled to append the output separators before

terminating the transaction.

The write device data subfunction contains only 2 options. The first is to decide whether to append the output separators to the final transmit record for the request. The second is to toggle the handshake bit.

13.4 The Frontplane

The frontplane handles all interactions from the card to the devices. This include transmitting and receiving data, speed sensing, and other special conditions generated by the SIO.

13.4.1 EXT_ISR (&EXISR)

The external status interrupt service routine only processes break detection. The break detection occur in two stages. The first stage is the start of break, and the last stage is the end of break. The firmware will set the flag for the port at the start of break. On the next interrupt the firmware will know if this is the end of break by checking the flag.

The user have the option allowing the null character to be inserted into the receive buffer. The break event is not generated if the interrupt enable mask is not set for the break event. However, the break event in the status vector CARD_ST will be set.

13.4.2 RX_ISR (&RXISR)

The receiver interrupt service routine uses macro expansion per port to gain fast execution. The major draw back of doing this is the ROM space usage and the difficulty of debug the code.

13.4.2.1 RXISR (Macro)

This is the macro which is expanded per port to process the "normal" characters received from the SIO. Data interrupts from each SIO channel are vectored directly to the expanded macro for the port which interrupted. The machine state is saved (exchange with the alternate register set), and the character is read from the SIO. Note that errors (parity, framing, overrun, etc) are handled by a different ISR which is vectored to directly when the error happens. The character read is ANDed with the current parity mask which strips off un-used bits for the 5, 6, and 7 bit modes. The mask is "FF" in 8 bit mode, "7F" in 7 bit, "3F" in 6 bit, and "1F" in 5 bit.

The entry point at RX_ISnA is entered by the error handling ISR (SPISR) in the event of a parity error with the ignore parity errors option set.

In order to maintain the highest speed, each character is not checked for all of the various options (backspace, delete, echo, etc, etc, etc) on every interrupt. That would take too long. Instead, there is a table of 256 entrys (the Special Character Table) which is indexed into by the character value. Each entry contains bits which represent the attributes of that character. The attributes are: handshake, signal, quotable, edit, and single text terminator. A generic character has all bits zero.

To implement the attribute toggle feature of RDD, a mask is kept which allows certain of these bits to be masked (turned off). A masked-out bit is equivalent to a bit which is not set, i.e. the special nature of that character

is limited. Each received character is used as an index into this table, the entry in the table is masked with the TOGL_MSK, and the result tested for zero. If it is zero, the character is not considered special, and it is placed in the buffer with no further checking. If the result was not zero, there is something special about the character, and a jump is taken to common code (outside the macro) to test which attribute(s) are indicated.

Putting the character in the buffer decrements the Frontplane Down Counter, which was set by SET_CNTR to the shortest buffer length which did not need additional processing (END-ON-something, XOFF handshake, etc). If the counter rolls over, an exit (jmp) is taken out of the macro to find out what it was that the counter was set to trigger on.

After the character is placed in the buffer the word "RX_FLAGS" is tested for zero. Any non-zero bit indicates that some further special processing is needed before the ISR can end. One of these is echo, another is alert-1.

The RX_BFULL bit in RX_FLAGS is checked after checking for special-ness via the special character table to allow handshake characters to be processed even when full.

13.4.2.2 RX_SPECL

This is where control transfers out of the macro if a character is received which has some non-zero bits in its entry in the special character table. In order of priority, the bits are checked for being handshake, signal, quotable, edit, or terminator characters. Once the class of character is determined, the appropriate routine is invoked to find out which character it is (there are several

different handshake and edit characters) and perform the appropriate function. Before leaving the macro, the IY register is loaded with the address of that port's stuff.

13.4.2.3 RX_EXIT

This is where all exits from the ISR go through, except those within the macro itself. Note that IY is popped from the stack.

13.4.2.4 RX_OTHER

This is where control is passed out of the macro in the event that there are some bits set in RX_FLAGS. The A register is assumed to have the RX_FLAGS contents for the first test of this routine. The rest (RX_OTHR2,3) use IY+RX_FLAGS since they are entered from the common code after a special character or frontplane counter rollover.

13.4.2.5 RX_ECHO

This routine echos the character in the E register.

13.4.2.6 RX_AL1

This routine generates the ALERT-1 event to the host, if it is enabled. The frontplane alert-1 bit in RX_FLAGS is reset so that this routine will only be called once per record (MX_RDD turns the flag back on) so the host will get one interrupt per record, and to speed processing.

13.4.2.7 RX_HSHK

We get here if the character received is identified as a handshake character. The character is compared to the current values of the device XON and XOFF, and the host ACK characters. If a match is found, and the indicated action is enabled, then the action is performed. The check for having the action enabled prevents incorrect operation if two characters are programmed to the same value with one of them turned off.

13.4.2.8 RX_QUOTE

This routine is entered whenever a character is received which is quotable, i.e. a character which may be preceeded by a "\" to turn off its "charm". These characters are the backspace, line delete, and quotable single text terminator. The previous character is checked for the "\" charcter, and if it is, the received character is layed over the "\". This is only done if the BS_PTR indicates that we did not backspace to this position. If we did, then the quotable character would be mis-interpreted as having followed the quote character.

13.4.2.9 RX_STT

This is where control is passed if the received character is one of the single text termination characters. If enabled, the sequence "CR", "LF" is echoed back to the terminal, and the received character is placed in the buffer (unless the strip STT option is set).

RX_STT5 is where the information is set to call PACKITUP and RX_COMPL to terminate the record and tell the host about it. RX_ENDIT is called from RX_COUNT and the error handling routine in SPISR.

The check for space prevents a record termination, which takes atleast 3 bytes for the header, from wrapping around the receive buffer. The check is made faster by checking for a backplane buffer, which usually isn't there. If it's not, then there can't be less than 8 bytes left since there is 512 bytes of space and the longest record is 252 bytes + 3 bytes for its header.

The swap of text termination parameters is then made, if there are any to swap.

13.4.2.10 RX_COUNT

This routine processes the characters which decremented the frontplane down counter to zero. First, the character is echoed, if enabled, since this wasn't done before the macro was exited. A check is made (via routine CHEK_XOF) to see if it is time to send an XOFF. Since this can happen at the same time one of the other counters hits zero, further checks are made.

The end-on-count down counter is updated, and tested for zero. The host buffer full counter is decremented next, followed by a check for the internal buffer size of 252. If any of these counters are exhausted the buffer is terminated with the appropriate termination code. The first condition to cause termination is used, and the rest of the checks are skipped.

The space remaining in the receive buffer is then calculated. If it is less than 8 bytes the receive buffer full flag is set to prevent any further chanracters from being received, and the current record is terminated with the buffer overflow code.

A new value of the frontplane down counter is calculated and installed, and one last check is made, this one for alert-1 mode.

13.4.2.11 RX_EDIT

This is where backspace and line delete characters come. If the received character gets here and it is neither of the edit characters, an internal bug has appeared, so we jump to die to alert someone's attention.

RX_DEL processes the line delete character. The pointers to the active receive buffer are re-set to point to just the header (an empty buffer), and the backspace pointer is re-set to the address of the header (an impossible place to backspace to). The header is cleared of any error flags, and the frontplane down counter is re-calculated. If echo is on, a "\ cr lf" sequence is echoed to the terminal.

RX_BACK processes the backspace character. If there is nothing in the receive buffer, a backspace has no effect. The active receive buffer's next character pointer is decremented, and the character deleted saved incase the echo mode is set to echo a "\" and the character. The echo mode is checked, and if echo is enabled, the appropriate sequence of characters is echoed to the terminal.

The position of this backspace is saved in BS_PTR so that the quote routine knows whether a quote character was backspaced to or not.

13.4.2.12 RX_SIGNL

This processes characters which are signal characters. Signal characters are like the BREAK condition; they cause an interrupt to the host and are tossed away. The appropriate interrupt code is found by looking for which of the 4 possible signal characters was received. The interrupt bit is then set and an event generated, if there wasn't one already set.

13.4.2.13 PACKITUP

This routine packs up the current receive buffer. The header is set to reflect the length of the buffer, and the terminating condition and character. The receive buffer is then set up for the next record.

13.4.2.14 PUT_CHR

This routine puts a character in the active receive buffer. The condition code is left set as appropriate for a check for buffer full. The character to be stored comes from the E register.

13.4.2.15 SET_CNTR

This routine sets the frontplane down counter. The value of the down counter is the smallest buffer which does not have any special processing to be done because of count. For example, if the host read was posted for 500 bytes, and the internal buffer size is 252, and the end on count is 50, but there are only 30 bytes remaining in the port's receive buffer, then the down counter would be set to 30.

The code is designed to be as fast as possible based on "usual" conditions. Usually, there is not a backplane record, so there's lots of space left (hence no XOFF possible), and end on count is not enabled.

13.4.2.16 MIN

This routine returns in HL the smaller of the two 16 bit unsigned numbers in DE and HL.

13.4.2.17 AL1_EVNT

This routine generates an alert-1 event if events are enabled. The event block is filled in and the event manager is called.

13.4.2.18 CHEK_XOF

This routine checks to see if it is time to send an XOFF, and if so, sends it. An XOFF is sent if there is less than 72 or so bytes of space remaining. This gives at least room for 16 or so characters in the worst case where end on count is set to 1.

RX_HXOF3 (this stuff used to be in the RXISR) checks the return address for the RX_COUNT routine. If it was called from there the frontplane down counter is not updated since that would wipe out all traces of the information which RX_COUNT needs. RX_COUNT therefore calls SET_CNTR itself.

13.4.3 SPC_ISR (&SPISR)

The special condition interrupt service routine handles all the error conditions produce by the SIO. These error conditions include the data overrun, framing, and parity error. The ISR has several options for processing the error condition.

If no options are specified, an error condition will cause the termination of the current receive frontplane record. If the ignore all errors option is set, then the error condition will be

ignored; that is, the character will be thrown away with no processing. If the ignore parity option is set, the parity error condition will be ignored and the received character will be processed as if there was no error. There is one additional option not to terminate the record but to replace the bad character was a user specified replacement character. For additional details and the logic flow, see the ERS.

13.4.4 SPD_SEN (&SPDSN)

This subprogram along with the speed sensing interrupt service routine, which will replace the normal receiver ISR when speed sensing is enabled, will handle all the speed sensing detection. See the chapter on speed sensing for additional details.

13.4.5 SS_ISR (&SSBIR)

This is the speed sensing interrupt service routine to replace the normal receive interrupt service routine when the speed sensing is enabled. See the chapter on speed sensing for additional details.

13.4.6 SSB_ISR (&SSBIR)

This is the service routine to replace the normal special condition interrupt service routine when speed sensing is enabled. When speed sensing is enabled the SIO can data overrun in synchronous mode when the ISR cannot read the characters out of the SIO buffer fast enough. Under this condition speed sensing information has been lost, and the SIO should be resynchronize.

The data overrun condition can occurred when all ports are doing speed sensing at the same time or when many ports are terminating a record at the same time. Hopefully this should not occur too often to cause user complaint.

13.4.7 TX_ISR (&TXISR)

The transmitter is like the receiver ISR in the sense macro expansion is used to speed up the processing. The normal path where the character is read from the SIO and put into the buffer is given the fastest execution speed by using straight line coding.

The ENQ/ACK counter is decrement after each character is transmitted. If the counter should count down to zero, a flag is set to transmit the ENQ on the next transmitter interrupt.

Some character or option checking are alway done even though the

option is not enabled. This is done to save processing time. For example, the transmitted character is always checked to see if it is a record separator. For majority of the case it is not. This save time because the byte compare is faster then the bit compare to see if the option is enabled.

13.5 The Self-test (MXSTEST)

Most of the documentation for the MUX self-test is contained in the listing. The self-test wa leverage from the ASI firmware which was originally from the MEF PSI.

The changes from the ASI include the following:

- o The hood sensing is different. The SIO modem control lines are used instead of a separate register on the card.
- o The ROM test will check 16 Kbytes.
- o A short RAM test is performed first before doing the BIC test. The test only checks the locations that are going to be used by the BIC test. The BIC test is done first because the RAM test takes so much time, and the host can set the BIC PA bit to cause the BIC self-test to fail.
- o The RAM test has been changed to test the 48Kbytes of the 64K RAM chip. The other 16K cannot be tested because of the MIC limitation. (The untested portion will never be access because it is overlay by the ROM address space.)
- o The RAM test is more extensive then the ASI. The short test is the same as the ASI; but the long RAM test uses a different test pattern and actually execute instructions out of the RAM space.
- o The number of CTC tested has been increased. No interrupt test is performed on the last 2 CTC because it is not connected to the interrupt chain.
- o The number of SIO tested has been increased. In addition 3 passes through the SIO test is performed for each port. The first pass is the internal loopback. The next 2 passes are performed if the loopback hood is installed. They will check the single-ended drivers and the differential drivers.
- o The MIC test checks both channel A and channel B DMA.
- o Code has been added to the self-test to determine whether a SIO has been damaged because of incorrect installation of the

cable. When the cable is installed incorrectly, it will probably kill one of the SIO. The damaged SIO will generate interrupt infinitely thus preventing the firmware from executing any code except the ISR servicing the interrupt.

The added code will enable the interrupt system. If it is able to reach the point of diabling the interrupt system, then there are no devices on the card generating infinite interrupt.

13.6 Miscellaneous

The following subprograms are essential for the proper operation of the firmware but does not fall into the above catogeries.

13.6.1 DMAA_ISR (&DMAA)

The only purpose of this subprogram is to process MIC channel A DMA interrupt which should never occur. A bug in the BIC would cause the MIC to generate this interrupt unnecessarily. A counter was included to note how often this occur.

13.6.2 MUXIVEC (&MUXIV)

This subprogram contains all the interrupt vectors for the SIO , CTC, and MIC devices. The tables are loaded into RAM during the initialization process.

The reason RAM is used instead of ROM is that the SIO interrupt vectors can be changed depending on whether speed sensing is enabled or not for a particular port.

The reason why all of the interrupt vectors are in RAM even those that does not change is because the I register can only be set to one value.

13.6.3 MUXMAIN (&MUXMN)

This is the main program of the MUX firmware. Upon any reset condition, the CPU will start instruction execution from address 0.

This subprogram will initialize the firmware system such as the SIO, CTC, RAM, BIC, MIC, etc before dropping into the idle loop.

13.6.4 MUXVAR (&MUXVR)

This subprogram contains all the global variable definitions.

13.6.5 REALCLK (&MXCLK)

This subprogram processes all the CTC #0 channel 3 interrupts which occur every 10 milliseconds. The subprogram will scan through the timers to see if anything needs to be done. For more details, see chapter 9 on the Timer Algorithms.

ZILOG SIO NOTES — CHAPTER 14

The following are notes concerning the Zilog SIO.

- SIO interrupt priority

```
high  1. channel A receive or
 |       special condition
 |    2. channel A transmit
 |    3. channel A external/status
 |    4. channel B receive or
 |       special condition
 v    5. channel B transmit
low   6. channel B external/status
```

- SIO interrupt vector

```
XXXX YYY0 channel B transmit
XXXX YYY2 channel B external/status
XXXX YYY4 channel B receive
XXXX YYY6 channel B special receive condition

XXXX YYY8 channel A transmit
XXXX YYYA channel A external/status
XXXX YYYC channel A receive
XXXX YYYE channel A special receive condition
```

- The SIO is programmed by the firmware on power up and whenever any parameters related to the SIO is changed.

- When a framing error occurred, the SIO will interrupt to the special receive condition vector even when the "parity does not affect vector" is programmed into write register 1. The manual is miss leading in specifying "or on special condition."

- Once a break condition is detected, the firmware must reset the external status and wait for the termination of the break condition. After the break is terminated, the SIO will generate another external status interrupt. Reset the external status again so that it can detect the next break.

- A null character is always generated after the break condition is terminated. In theory the receive interrupt should be generated first before the external status interrupt since the

former has a higher priority. In practice this is not always true because of a bug in the SIO. The MUX firmware will turn off the receive interrupt when a break is received. After the break is terminated, the null character will be read and discarded before the receive interrupt is turned back on. However, if "general null" option is enabled, the null character will not be read and discraded.

- If the SIO is programmed for odd parity, the null character generated by the SIO for the break detection will generate a parity error. This must be cleared before reading the null character.

- The SIO is programmed by subprogram SET_SIO.

- Whenever the SIO is reporgrammed for whatever reasons, the SIO is not reset again after the power-up initialization sequence. If the SIO is reset, the interrupt vector will be lost. This will require more firmware to program the SIO instead of using one common routine for every port.

- A SIO can generate infinite interrupts continuously if the modem signal input is damaged. This will prevent normal execution of the firmware.

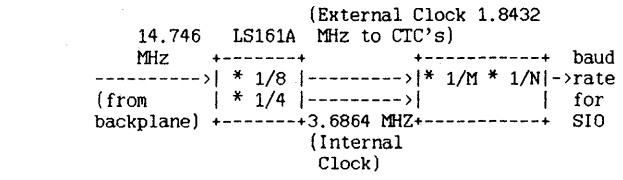
- The SIO transmitter must be disabled first before the SIO is reprogrammed for synchonous "hunt mode" for speed sensing. If this is not done a garbage character will be sent to a terminal.

- The SIO external/status and the special condition channel must be reset after speed sensing is completes successfully and the SIO is reprogrammed for the asynchronous mode. In addition the receiver buffer must be emptied by reading the characters from the SIO.

- The speed sense receive ISR must check the external/status hunt mode bit to see if the incoming character is valid. There may be characters remaining in the SIO receive buffer after the SIO is reprogrammed for the synchronous hunt mode after a failure. The invalid characters are from the previous speed sense.

ZILOG CTC NOTES	CHAPTER 15

o The following is the block diagram of the hardware for generating the baud rate to CTC channel 0.



where M=Prescaler Of The CTC and
N=Time Constant Register of the CTC

o The following is the M and N settings for generating the supported baud rates.

Count Source	M	N	BAUD Rate	Error*
From external clock (1.8432 MHz) to CLK/TRG0 of the CTC	X	3	38400	
	X	6	19200	
	X	12	9600	
	X	16	7200	
	X	24	4800	
	X	32	3600	
	X	48	2400	
	X	64	1800	
	X	96	1200	
	X	128	900	
	X	192	600	
From the internal clock (3.6864 MHz) of the CTC	16	48	300	
	16	96	150	
	16	107	134.5	0.06%
	16	131	110	0.07%
	16	192	75	
	256	18	50	

* No Error Unless Otherwise Noted

Where M=Prescaler Of CTC,
N=Time Constant Register of CTC,
X=Don't Care.

o There are 3 CTC's on the 8-channel MUX card. They are used as follow:

```
CTC #0 Ch 0 - DMA pacer
       Ch 1 - port 1 baud rate generator
       Ch 2 - port 0 baud rate generator
       Ch 3 - firmware real time clock

CTC #1 Ch 0 - port 2 baud rate generator
       Ch 1 - port 3 baud rate generator
       Ch 2 - port 4 baud rate generator
       Ch 3 - not used

CTC #2 Ch 0 - port 5 baud rate generator
       Ch 1 - port 6 baud rate generator
       Ch 2 - port 7 baud rate generator
       Ch 3 - not used
```

- o All the CTC's programmed as baud rate generator should never generate an interrupt.

- o All the baud rate generators are programmed by subprogram PRG_CTC.

- o The real time clock is programmed to generate an interrupt every 10 milliseconds. This is done in subprogram MUX_MAIN when the firmware environment is initialized.

- o The DMA pacer is programmed to generate a pulse to the MIC for pacing. This is done in subprogram MUX_MAIN when the firmware environment is initialized.

ADDITIONAL NOTES	CHAPTER 16

- o Due to the hardware nature of the Z80 on fetching data, the 16-bit quantity from the host must have the high and low byte swapped before being used by the Z80 as a 16-bit quantity. This comes about as follow: the host will transfer the high byte of a 16-bit quantity first to the card. The card Z80 will store the byte in low RAM address and the next byte of the 16-bit quantity in the next higher RAM address. When the Z80 fetch the 16-bit quantity, the host high byte will be the Z80 low byte. Thus, the bytes must be swap.

- o The frontplane interface subprograms must swap the high and low bytes of all 16-bit quantity being returned to the host.

- o When an interrupt occurred, the interrupt system is not reenabled until the processing for the interrupt is completed. The only exception to this is the BIC/MIC interrupt.

- o The alternate register set is used by all the interrupt service routines except for the BIC/MIC interrupt. The BIC/MIC interrupt service routine will use the current register set after pushing its contents onto the stack.

- o Register IX contains the address to the status vector byte STATUS. This register is used extensively by the macros SBIT, POST, TEST, CLEAR, FBIT, FSET, FTST, and FCLR to define, to set, to test, and to clear a status vector bit or a flag bit.

- o Register IY contains the address to the transaction table for the current transaction being processed by the backplane.

- o Note that the I/O port address does not offer full address decoding. Therefore, more then one different address may be used to access the device. This may be a problem in debugging strange behavior of the card. Like writing to I/O port address 0 may cause the SIO or CTC to do something because the decoding uses only one line of the address to select the device.

ASCII CHARACTERS & BINARY CODES — APPENDIX A

	0	1	2	3	4	5	6	7
0	NUL	DLE	sp	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

EIA RS-232-C CONNECTOR PIN ASSIGNMENT — APPENDIX B

pin no.	description	CCITT V.24 Equivalent	EIA RS-449 Equivalent
1	AA Protective Ground	101	
2	BA Transmitted Data Send Data	103	SD
3	BB Received Data	104	RD
4	CA Request to Send	105	RS
5	CB Clear to Send Ready for Sending	106	CS
6	CC Data Set Ready Data Mode	107	DM
7	AB Signal Ground (Common Return)	102	SG
8	CF Received Line Signal Detector Data Channel Received Line Signal Detector Receiver Ready	109	RR
9	(Data Set Testing)		
10	(Data Set Testing)		
11	Unassigned		
12	SCF Secondary Received Line Signal Detector Backward Channel Received Line Signal Detector Secondary Receiver Ready	122	SRR
13	SCB Secondary Clear to Send Backward Channel Ready	121	SCS
14	SBA Secondary Transmitted Data		

		Transmitted Backward Channel Data Secondary Send Data Not supported by firmware	118	SSD
15	DB	Transmission Signal Element Timing (DCE Source) Send Timing Not available on this card	114	ST
16	SBB	Secondary Received Data Received Backward Channel Data Not supported by firmware	119	SRD
17	DD	Receiver Signal Element Timing (DCE Source) Receive Timing Not available on this card	115	RT
18		Unassigned		
19	SCA	Secondary Request to Send Transmit Backward Channel Line Signal	120	SRS
20	CD	Data Terminal Ready Terminal Ready	108.2	TR
21	CG	Signal Quality Detector Data Signal Quality Detector Signal Quality Not available on this card	110	SQ
22	CE	Ring Indicator Calling Indicator Incoming Call	125	IC
23	CH/CI	Data Signal Rate Selector (DTE/DCE Source) Signaling Rate Selector Signaling Rate Indicator	111/112	SR SI
24	DA	Transmit Signal Element Timing (DTE Source) Terminal Timing	113	TT

		Not available on this card		
25		Not defined		

The following is a historical note on the EIA RS-232-C symbol name. The original pin definition was defined by the pin assignment which consists of a matrix of 3 rows and 6 columns giving a total of 18 pins, see below.

	A	B	C	D	E	F
A	.	.	.	.	.	.
B	.	.	.	.	.	.
C	.	.	.	.	.	.

Pin AA was used for protective ground, pin AB was for signal ground, and so on. The pin assignment was carried over to the current EIA RS-232-C symbol definition.

Table of Contents

HP-CIO 8-CHANNEL MUX FIRMWARE IMS