NCAR/RAF VMEbus Interface for the
Particle Measuring Systems Optical
Array Grey Probe

Jerry V. Pelk
TABLE OF CONTENTS

List of Figures .......................................... iii
Preface ................................................. v
Acknowledgements ...................................... vii
Abstract ................................................ ix
1. Introduction ....................................... 1
2. General Circuit Description ..................... 3
3. Probe Data Unloading ............................. 5
4. Particle Timing and Statistics .................. 9
5. Charge Sampling ................................... 11
6. VME Bus Interface ............................... 13
Appendix A ............................................ 15
Appendix B ............................................ 21
Appendix C ............................................ 25
Appendix D ............................................ 41
LIST OF FIGURES

Figure 2.1 VME Grey Scale Interface Block Diagram. ............. 3
Figure 3.1 Probe Connector Diagram. ........................... 5
This Technical Note describes the design and operation of the NCAR/RAF VMEbus interface board for the Optical Array Grey Probe manufactured by Particle Measuring Systems, Inc. of Boulder, Colorado. This interface was developed at the NCAR Research Aviation Facility in order to provide an interface between the Grey Probes and the Aircraft Data Systems flown on the RAF aircraft. The RAF Aircraft Data Systems are designed around the VMEbus which is a computer bus widely used for industrial and instrumentation computers. There were no commercial interface boards available which met our needs, so a custom board has been developed.

This Technical Note does not attempt to describe the operation or design of the Grey Probe other than the data and interface signals. It is assumed the reader has a basic understanding of the Grey Probe and its interface signals. The reader is referred to the manuals available from Particle Measuring Systems for more information about the Grey Probe.
ACKNOWLEDGEMENTS

The author wishes to express his appreciation to the many people who provided advice and assisted in the development of this project. Mike Spowart and Kim Weaver provided valuable technical advice for development and review of the design. Herminio Avila supplied his superior CAD skills in generating the board layout. Kurt Zrubek spent many hours tracking down and ordering components. Hung Ta did an excellent job assembling the prototype board, and making necessary changes as the design evolved.
ABSTRACT

The Optical Array Grey Probe manufactured by Particle Measuring Systems of Boulder, CO can be used for the study of cloud water droplets and ice particles. The Grey Probe captures a 2-dimensional grey scale image of particles passing through its sampling volume. It was desired to fly the Grey Probes on the NCAR Research Aviation Facility aircraft, but a way was needed to interface the probe to the Aircraft Data Systems. An intelligent VMEbus interface board has been developed by the RAF Instrumentation Group to provide this interface for two Grey Probes.

One of the Grey Probes operated on the RAF sailplane has been modified to make analog charge measurements of a particle as it passes through the probe. The interface was therefore designed to sample and digitize these charge measurements as well as collect the image data from the probe.

This document describes the design and operation of the VMEbus Grey Probe Interface. Schematic diagrams, DSP source code listings, and Programmable Logic Device source code listings are included.
1. INTRODUCTION

The NCAR/RAF VME Grey Scale interface provides a high performance, intelligent interface to a VME bus based data collection system for up to two Particle Measuring Systems (PMS) 2D Optical Array Grey Probes. The interface may be configured under software control for both the GA1 and GA2 model probes. The interface is built to the VME bus 6U form factor.

The interface also provides charge and splash analog inputs. These allow sampling of probes which have been modified, according to an NCAR design, to make particle charge measurements.

The card is easily interfaced to a VME system via a dual-port memory. The sampling of the probes is controlled by the host processor by writing to defined control structures in the dual-port memory. An Analog Devices ADSP-2101 Digital Signal Processor on the interface collects data from the probes, computes particle statistics, and places data into defined buffers in the dual-port memory. The host VME processor reads the data from the dual-port memory across the VME bus. The interface may also be software configured to generate a VME bus interrupt when a buffer of data has been acquired.
2. GENERAL CIRCUIT DESCRIPTION

Figure 2.1 shows a block diagram of the Grey Scale interface. Appendix A contains a circuit diagram for the interface. Bus transceivers provide buffering of the VME address, data, interrupt, DTACK* and WRITE* lines. Address decoding, and the VME data transfer bus interface are provided by an Advanced Micro Devices MACH210 programmable logic device (PLD). Another MACH210 PLD provides programmable VME bus interrupt capability.
An 8K x 16 dual port memory is used to hold grey scale image and charge data blocks until they are read by the host processor. Control structures are also defined in the dual-port memory, through which the host processor may control the sampling of the probes. Access to the dual-port memory by the host VME processor and the on card DSP is controlled through hardware semaphores provided by the dual-port memory chips.

The heart of the interface is an Analog Devices ADSP-2101 integer DSP. The DSP runs at a clock rate of 12.5 Mhz. It features 2K words of 24 bit internal program memory, and 1K words of 16 bit internal data memory. 2K words of 24 bit external program memory are also provided on the card. The ADSP-2101 is capable of single clock instruction cycles. Two highly configurable serial ports are also provided. These serial ports are used to acquire data from the grey scale probes.

Upon reset, the ADSP-2101 boots from a 16K x 8 EPROM. The sampling program is loaded from EPROM into internal and external program memory. The sampling program then begins execution. The ADSP-2101 communicates with the rest of the interface via local 14 bit address and 16 bit data buses. Local address decoding is provided by a MACH210 PLD. The dual-port memory and other interface devices are accessed by the DSP across the local bus with 1 wait state.

A programmable true air speed clock is provided by an Analog Devices AD7245A 12-bit D/A converter, a Burr-Brown VFC110 voltage-to-frequency converter, and a 74HC4046 phase-locked loop.

Particle charge sampling is provided by an Analog Devices AD7874 4 channel, 12 bit, 100 Khz A/D converter. This A/D features 4 sample-and-holds. When a convst* signal is issued, the sample-and-holds acquire the analog signals. The A/D converter converts each of the 4 signals and stores the results in an internal FIFO. An interrupt request is then issued to the DSP, which reads the data from the FIFO across the local data bus. The A/D converts at a 100 Khz rate. With interrupt latency the 4 charge channels may be sampled at up to a 20 Khz rate.

Charge analog signal conditioning is provided by Burr-Brown INA117 instrumentation amplifiers, and Frequency Devices D70L6 6-pole Butterworth low-pass anti-aliasing filters. The -3db corner frequency of the filters is set at 2500 Hz.
3. PROBE DATA UNLOADING

The ADSP-2101 provides two flexible serial ports (SPORTs). These ports along with a MACH210 PLD are used to control and unload data from the Grey Scale probes. The probes are connected to the interface via two 25 pin D microminiature connectors. These connectors are mounted on the interface faceplate, and are labeled Probe 1 and Probe 2. Figure 3.1 shows the pin connections for these connectors.

Figure 3.1 Probe Connector Diagram.

Appendix B shows timing diagrams for the probe data unloading signals. The signals to and from each probe are as follows:
Probe Control (PCTLx). A 16 bit control word is sent to the probe during the End-Of-Particle period.

End-Of-Particle (EOPx). This signal is asserted when the interface has detected the end of a particle in an image. This signal resets the probe for acquisition of another particle.

Unload (UNLDx). This signal causes the probe to move one 16 byte slice from its memory to its shift registers. This prepares the slice to be serially shifted down the data lines to the interface.

Bit Shift (BSHFTx). This signal clocks the probe shift registers and causes the data to be serially transmitted to the interface. The Bit Shift signal also clocks the probe control byte sent from the interface to the probe.

True Air Speed Clock (TASx). This signal is used by the probe to control the timing of the image slices while acquiring a particle image.

Full (FULLx). This signal is asserted by the probe to indicate that its memory buffer is full, and a particle image has been acquired.

Data (DATAx). The serial data sent from the probe to the interface. Data are transmitted in 16 byte slices.

The probe control word is generated by writing the word to the SPORT. The SPORT DT line is connected directly to the PCTL line driver.

The EOP signal is generated by the SPORT TFS line. The TFS line is programmed to be asserted continuously during a write operation. Thus while the probe control word is being sent to the probe, the TFS line is asserted, and an EOP signal is generated.

The Unload signal is generated from the SPORT RFS line through the Probe Unloading PLD. The RFS line is programmed in the DSP to be asserted every 64 bits read. The RFS signal is sent to the PLD, which passes every other RFS pulse through as the UNLD signal. Thus an UNLD is generated at the end of each 128 bit slice read. Upon receipt of the UNLD signal, the Grey Probe loads its output shift registers with the next slice of image data. It should also be noted that UNLD is asserted at the rising edge of SCLK, and the SPORT latches data on the falling edge of SCLK. This 1/2 clock period allows time for the probe to latch a data slice into its shift registers, and drive the data line with the value of the first bit of the slice.

The Bitshift clock is generated from the SCLK signal of the SPORT. The SCLK signal is sent to the Probe Unloading PLD where it is inverted and sent out as the Bitshift clock. SCLK runs at 2.0833 Mhz.
The True Air Speed clock is created when the AD7245A D/A produces a 0-10V output corresponding to a 12-bit word written to it by the DSP across the local bus. The D/A 0-10V output drives the VFC110 input, which produces a corresponding 0-3 Mhz output. The VFC110 output feeds the 74HC4046 which multiplies the frequency by 4 to produce a 0-12 Mhz tasbase signal. This tasbase signal is then divided by programmable dividers incorporated into the probe control MACH210 PLD to produce separate TAS clocks for each probe. The tasbase signal is divided by 2, 4, 8, or 16, according to the probe optical resolution. The tasbase dividers also provide the phase-locked loop with the tasbase divided-by-4 comparator signal.

The Full signals from the probes are read by the DSP through the Probe Unloading PLD. The Probe 0 Full line is read on DO, and the Probe 1 Full signal is read on D1 when RDFULL is asserted by the DSP. A DSP interrupt request is generated on the rising edge of the Full line from either probe.

Data are read directly from the probe by the SPORT. The probe shifts a bit of data on the rising edge of each BSHFT cycle, and the SPORT reads the bit on the falling edge of each BSHFT cycle. During an UNLD pulse, BSHFT is suppressed. This allows the first bit of a slice to be read prior to the second bit being shifted out.
4. **PARTICLE TIMING AND STATISTICS**

The global control struct, described in the VME Bus Interface section below, includes fields for initializing the interface time. The DSP keeps an internal real-time clock in hours, minutes, seconds and milliseconds. The DSP also keeps counters which count the number of seconds and microseconds that have elapsed since the last power on or software reset. The microsecond counter has a resolution of 2 microseconds. In addition to particle timing, the DSP computes various statistics for each particle. These include particle length, width. The statistics are stored in the particle image header.

A probe is enabled to acquire a particle by the assertion of the End-Of-Particle (EOP) signal. When the DSP enables a probe to acquire a particle by asserting the EOP signal, the particle header structure is stamped with the current second and microsecond counts. When a particle is acquired by the probe, indicated by the Full line being asserted, the particle header is again stamped with the current second and microsecond counts, as well as the current hour, minute, second, and millisecond time. The difference between probe enabled and particle acquired counter values precisely measures the probe 'On' time prior to acquiring a particle. The counter values may be used for calculating inter-particle times and particle densities. The number of particles detected by the probe while not enabled is also stored in each particle header. The real-time clock time-stamp of each particle allows grey scale data to be correlated to other sampled data being taken by the system.
5. **Charge Sampling**

The Grey Scale interface provides charge and splash analog input channels, to sample charge data from probes which have been modified to make charge measurements. The AD7874 operates in a continuous sampling mode, taking charge measurements at a rate dependent on the true air speed and the desired charge sample spacing. The probe control structures provide fields for programming the desired charge sample spacing. When a particle image is acquired by the probe, 32 samples of charge and splash data are stored in the dual-port memory data buffer following the particle's image data.

The AD7874 convst* signal is generated by a programmable divider incorporated into the address decoder PLD. The convst* divider is a 10 bit down counter which divides the tasbase signal. The upper 8 bits of the counter are programmable. The counter is programmed by the DSP to provide the desired charge sample spacing as a particle moves through the charge detection rings. At 120 m/s true air speed, a charge sample spacing of .6 cm may be achieved. At 50 m/s, the charge sample spacing may be as low as .25 cm.

The DSP maintains a circular buffer of 64 charge measurements for each analog channel. When a particle is detected in a probe by the FULL line of the probe being asserted, the current position of the circular buffer is noted. The DSP maintains a computed number of samples that will have been taken from when the particle was centered in the charge ring to when the FULL line was asserted. Enough additional samples are then taken, such that a 32 sample window is acquired with the window center corresponding to the particle being centered in the charge ring. These 32 samples of charge and splash data are stored following the particle's image data.
6. VME Bus INTERFACE

The Grey Scale interface is programmed by the bus interface PLD to appear on the VME bus at a base address of 0x600000 or 0x610000 of the A24:D16 address space. Jumper H1 determines which address is used. If H1 is installed the base address is 0x600000. If H1 is removed, the base address is 0x610000.

Following is a memory map of the interface as seen from the VMEbus. These addresses should not be confused with those listed in the vminegrey.h file listed in Appendix D. Those addresses are for the dual-port memory as seen by the DSP.

base + 0x0000    probe 0 control struct
base + 0x0040    probe 1 control struct
base + 0x0060    global control struct
base + 0x0100 to 0x1FFF    probe 0 data buffer
base + 0x2000 to 0x3FFF    probe 1 data buffer
dual-port semaphores
base + 0x4000 to 0x400F
base + 0x8000
base + 0x8002
base + 0xC000

The probe control structures are defined as follows:

struct GreyPctl {  // individual probe control struct
    long id;  // probe id, "G1", "G2"
    short type;  // 0 = none, 1 = GA1, 2 = GA2
    short resolution;  // resolution (microns), 25,50,100,200
    short chg_spc;  // charge spacing (mm), 0 = none
    short chg_loc;  // charge ring distance from beam (mm)
    short control;  // control byte to send to the probe
    short spare;
};

struct GreyGctl {  // global control struct
    short full[2];  // buffer full flags
    short hour;  // set interface hour
    short minute;  // set interface minute
    short sec;  // set interface second
    short set_time;  // set time flag
    short tas  // current true air speed (m/s)
    short go;  // control structures initialized flag
};
The control structures are 'zeroed' out by the DSP shortly after a reset. VME bus interrupts are also disabled by a reset. The host VME processor initiates sampling of one or both probes. This is done by first programming the desired interrupt level and vector into the BIM. Next the host processor fills in the probe control structs, and sets the interface time and initial true air speed in the global control struct. The host processor then sets the go flag in the global control struct. Once a probe control struct has been properly initialized and the go flag has been set, the DSP will begin to monitor that probe for data.

When the data buffer has been filled, the DSP will set the full flag in the global control struct. If interrupts have been enabled, an interrupt will be issued to the host processor. The host responds by reading the data from the buffer, and clearing the full flag.

At 1 second intervals, the host processor writes the current true air speed into the global control struct.

Before the host processor can access the dual-port memory control structures, it must first acquire a semaphore by writing a zero to the first semaphore address. The host processor then reads the semaphore. If the semaphore returns a zero, the DSP has released the memory. The host processor may then read from or write to the control structures. The semaphore is then released by writing a 1 to the semaphore. This allows the DSP to gain access to the structures.
APPENDIX A

Schematic Diagrams
APPENDIX B

Timing Diagrams
DATA SLICE UNLOAD TIMING

EOP TIMING
Appendix C

Programmable Logic Device Source Listings
; PALASM Design Description

; --------------------------------- Declaration Segment ------------
TITLE VME Grey Scale Bus Interrupter.
PATTERN 0001
REVISION A
AUTHOR Jerry V. Pelk
COMPANY NCAR, Research Aviation Facility
DATE 09/27/93

CHIP _vgbim MACH210

; --------------------------------- PIN Declarations ---------------------
PIN 35 CLK1 comb ;input, clock
PIN 18 /RESET comb ;input, chip reset
PIN 32 CS comb ;input, chip select
PIN 31, 14 LIRQ[1..0] comb ;input, interrupt reqs
PIN 33, 13, 11 A[3..1] comb ;input, address lines
PIN 42, 3, 43, 27, 26, 25, 24, 2 D[7..0] comb ;i/o, data lines
PIN 10 /WRITE comb ;input, read/write
PIN 36 /AS comb ;input, data strobe
PIN 9 /DS0 comb ;input, data strobe 0
PIN 8 /IACK comb ;input, interrupt ack
PIN 5 /IACKIN comb ;input, iack daisy chain
PIN 4 /IACKOUT comb ;output, iack daisy chain
PIN 20, 21, 30, 29, 28, 40, 41 /IRQ[7..1] comb ;output, interrupt req
PIN 19 INTAE comb ;output, int ack enable
NODE 47 INTACK comb ;ack this interrupt
NODE 61, 29, 59, 13, 7, 25, 19, 63 VECTOR[7..0] reg ;vector register
NODE 31, 53, 15, 9, 27, 23, 11 CTRL[6..0] reg ;control register
NODE 41 DTRST comb ;data lines tri-state node

; Define the vector and control register select strings.

; --------------------------------- Boolean Equation Segment ------
EQUATIONS

; Define the vector register equations.
vector[7..0].clkf = clkkl
vector[7..0].setf = gnd
vector[7..0].rstf = reset
if (WV) then begin
   vector[7..0] = d[7..0]
end
else begin
   vector[7..0] = vector[7..0]
end

; Define the control register equations.
; ctrl[2..0] contains the interrupt level.
; ctrl[4..3] contain the interrupt enable bits.
; ctrl[6..5] contain the interrupt pending bits.
; An interrupt is cleared by writing a 0 into ctrl[6..5].

ctrl[6..0].clkf = clk1
ctrl[6..0].setf = gnd
ctrl[6..0].rstf = reset
if (WC) then
  begin
    ctrl[4..0] = d[4..0]
  end
else
  begin
    ctrl[4..0] = ctrl[4..0]
  end

; Define the interrupt request output lines.
case (ctrl[2..0])
begin
  0: begin
    irq[7..1] = gnd
  end
  1: begin
    irq[7..2] = gnd
  end
  2: begin
    irq[7..3,1] = gnd
  end
  3: begin
    irq[7..4,2..1] = gnd
  end
  4: begin
    irq[7..5,3..1] = gnd
  end
  5: begin
    irq[7..6,4..1] = gnd
  end
  6: begin
    irq[7,5..1] = gnd
  end
  7: begin
    irq[6..1] = gnd
  end
end

; Define the interrupt acknowledge enable equation.
intae = iackin * intack * ds0 ; put the vector on the bus
; Define the data lines output equations.
dtrst = intae + RC + RV
d[7..0].trst = dtrst
if (intae + RV) then begin ; interruptack or vector read
d[7..1] = vector[7..1]
d[0] = ctrl[6] ; which interrupt
end
else begin ; read control register
d[6..0] = ctrl[6..0]
d[7] = gnd
end

; Define the iackout equation.
iackout = iackin * /intack * as
; PALASM Design Description

;--------------------------------- Declaration Segment -------------

TITLE VME Grey Scale VME Bus Interface.
PATTERN 0001
REVISION A
AUTHOR Jerry V. Pelk
COMPANY NCAR, Research Aviation Facility
DATE 10/19/93

CHIP _vgbus MACH210

;--------------------------------- PIN Declarations -------------------

PIN 35 CLK comb ;input, clock
PIN 25 /SYSRST comb ;input, bus reset
PIN 42, 41, 40, 39, 38, 37, 36, 29, 30, 32 A[23..14] comb ;input, address lines
PIN 31, 19, 18, 17, 16 AM[5..3,1..0] comb ;input, address modifiers
PIN 15 /AS comb ;input, address strobe
PIN 13 /DS0 comb ;input, data strobe 0
PIN 26 /IACK comb ;input, interrupt ack
PIN 14 /LWORD comb ;input, long word access
PIN 27 /PORST comb ;input, power on reset
PIN 33 /WRITE comb ;input, read/write
PIN 28 BD_NUM comb ;input, board number, 0 or 1
PIN 10 INTAE comb ;input, interrupt ack input
PIN 20 /DTACK reg ;output, data transfer ack
PIN 24 /RESET comb ;output, board reset
PIN 43 /DPOE comb ;output, dual-port ram output enable
PIN 2 /DPCE comb ;output, dual-port ram chip enable
PIN 21 /DPSEM comb ;output, dual-port semaphore
PIN 3 /VME_ENAB comb ;output, bus tranceiver enable
PIN 4 BIM comb ;output, bus interrupter enable
NODE 17 BD_SEL comb ;board selected
NODE 3 SWRST comb ;software reset
NODE 49 WE comb ;write enable qualifier
NODE 65 DSHFT reg ;dtack delay register

; Set the board base address at 0x6y0000, where y is the board number
; 0 or 1. Respond to address modifier codes of 0x39 or 0x3D.

STRING VGBASE 'AS * /LWORD * /IACK *
*(A[16] :: BD_NUM)'

;---------------------------------- Boolean Equation Segment ------

EQUATIONS

;Define the board address equations.
bd_sel = VGBASE
vme_enab = bd_sel + intae ; tranceiver enab
dpce = bd_sel * /a[15] * /a[14] * we ; base+0x0000
dpsem = bd_sel * /a[15] * a[14] ; base+0x4000
bim = bd_sel * a[15] * /a[14] * we ; base+0x8000
swrst = bd_sel * a[15] * a[14] ; base+0xC000
;Define the board reset equations.
reset = swrst + sysrst + porst

;Define the dual-port ram output enable equation.
dpoe = /write

;Define the shift register and dtack equations.
; dtack is delayed one clock period from the time the board is selected
; and ds0 is asserted, or from the beginning of an interrupt ack
; interval.
dshft.clkf = clk
dshft.rstf = porst
dshft.setf = gnd
dshft = bd_sel + intae

dtack.clkf = clk
dtack.rstf = porst
dtack.setf = gnd
dtack = (bd_sel + intae) * (dshft + dtack)

;Define the write enable qualifier.
we = (write * ds0 * bd_sel * /dtack) + /write
;PALASM Design Description

;----------------------------- Declaration Segment ------------------
TITLE VME Grey Scale Adress Decoder and Charge Conversion Start
Control
PATTERN 0001
REVISION A
AUTHOR Jerry V. Pelk
COMPANY NCAR, Research Aviation Facility
DATE 10/04/93

CHIP _vgdecod MACH210
;---------------------- PIN Declarations -------------------
PIN 20 A0 COMB ; address lines
PIN 19 A1 COMB ;
PIN 16 A2 COMB ;
PIN 15 A3 COMB ;
PIN 11 A12 COMB ;
PIN 33 A13 COMB ;
PIN 14 /RD COMB ; read
PIN 3 /WR COMB ; write
PIN 32 /DMS COMB ; data memory select
PIN 4 /PMS COMB ; program memory select
PIN 43 /EXTPM COMB ; external program memory sel
PIN 42 /DPSEM COMB ; dual-port semaphore sel
PIN 26 /RDCHG COMB ; read charge data
PIN 30 /DMS COMB ; read full lines requests
PIN 29, 27 LIRQ[1..0] COMB ; interrupt requests
PIN 31 /SETTAS COMB ; set true air speed
PIN 24 PRGTAS0 COMB ; program tas #1 divider
PIN 25 PRGTAS1 COMB ; program tas #2 divider
NODE 39 PRGCHG COMB ; program charge divider

; Charge sampling divider pins and nodes.
PIN 10 /RESET COMB ; gen system reset
PIN 5, 6, 7, 8, 9, 21, 18, 17 D[7..0] REG ; data lines
PIN 35 CLK1 COMB ; chip clock
PIN 13 TASBASE COMB ; charge divider tas input
NODE 58, 56, 55, 53, 51, 13, 17 CDIV[9..2] REG ; charge div reg
NODE 12, 9, 6, 3, 30, 27, 24, 18, 14, 65 CCNT[9..0] REG ; charge ctr
PIN 2 /CONVST REG ; charge divider output

STRING DWAIT0 'DMS * (/A13 + A13 * /A12)' ; 0 wait state mem
STRING DWAIT1 'DMS * A13 * A12' ; 1 wait state mem

;----------------------------- Boolean Equation Segment -------
EQUATIONS
extpm = pms
dpen = DWAIT0 * /a13 ; dp mem enable
dpsem = DWAIT0 * a13 ; semaphore mem space
rdchchg = DWAIT1 * /a3 * /a2 * /a1 * /a0 * rd ; DWAIT1 + #h001
rdfull = DWAIT1 * /a3 * /a2 * /a1 * a0 * rd ; DWAIT1 + #h002
lirq[0] = DWAIT1 * /a3 * /a2 * a1 * /a0 * wr ; DWAIT1 + #h000
lirq[1] = DWAIT1 * /a3 * /a2 * a1 * a0 * wr ; DWAIT1 + #h003
;Load the charge divider register.
cdiv[9..2].clkf = clkl
cdiv[9..2].setf = gnd
cdiv[9..2].rstf = reset
if (prgchg) then
begin
  cdiv[9..2] = d[7..0]
end
else
begin
  cdiv[9..2] = cdiv[9..2]
end

;Charge divider equations. The charge divider is a down counter that counts down from the value in cdiv.
ccnt[9..0].clkf = tasbase
ccnt[9..0].setf = gnd
ccnt[9..0].rstf = reset
if (ccnt[9..0] = 0) then
begin
  ccnt[9..2] = cdiv[9..2]
  ccnt[1] = vcc
  ccnt[0] = vcc
end
else
begin
  ccnt[1] = ccnt[1] + ccnt[0]
  ccnt[0] = ccnt[0]
end

;A charge conversion start is generated when ccnt reaches 0.
convst.clkf = tasbase
convst.setf = gnd
convst.rstf = reset
if (ccnt[9..0] = 0) then
begin
convst = vcc
end
else
begin
  convst = gnd
end
;PALASM Design Description

;--------------------------------- Declaration Segment ------------
TITLE VME Grey Scale Probe Unloading Control and TAS control
PATTERN 0001
REVISION A
AUTHOR Jerry V. Pelk
COMPANY NCAR, Research Aviation Facility
DATE 11/03/93

CHIP _vgprobe MACH210

;--------------------------------- PIN Declarations ---------------
PIN 10 /RESET comb ; input
PIN 40 RDFULL comb ; input
PIN 35 CLK1 comb ; input
PIN 25, 43, 24 D[2..0] comb ; I/O
PIN 42 /DIRQ2 comb ; output
PIN 32 /CHGIRQ comb ; input

;Probe 0 pins and nodes.
PIN 39 SCLKO comb ; input
PIN 41 RFSO comb ; input
PIN 38 TFSO comb ; input
PIN 33 FULLO comb ; input
PIN 3 UNLDO comb ; output
PIN 2 BSHFTO comb ; output
NODE 9, 7, 5, 14, 13 CNTO[4..0] reg ;
NODE 54 RFSHO reg ;
NODE 43 RFSL0 reg ;
NODE 31 FULLO_CUR reg ;
NODE 30 FULLO_PREV reg ;
NODE 49 FULLO_INT comb ;

;Probe 1 pins and nodes.
PIN 18 SCLK1 comb ; input
PIN 11 RFS1 comb ; input
PIN 17 TFS1 comb ; input
PIN 19 FULL1 comb ; input
PIN 20 UNLD1 comb ; output
PIN 21 BSHFT1 comb ; output
NODE 25, 23, 21, 28, 26 CNT1[4..0] reg ;
NODE 53 RFSH1 reg ;
NODE 56 RFSL1 reg ;
NODE 32 FULL1_CUR reg ;
NODE 6 FULL1_PREV reg ;
NODE 65 FULL1_INT comb ;

;TAS divider pins and nodes.
PIN 13 TASBASE comb ; input
NODE 39, 37, 35, 15 TCNT[3..0] reg ;
PIN 36 PRGTAS0 comb ; input
NODE 10, 17 TDIVO[1..0] reg ;
PIN 27 TASO comb ; output
PIN 37 PRGTAS1 comb ; input
NODE 12, 51 TDIV1[1..0] reg ;

35
; Edge detector detects a falling rfs edge.
; Detect a high rfs level.
; Probe 0

rfsl0.clkf = clk1
rfsl0.setf = gnd
rfsl0.rstf = reset
rfsl0 = rfs0

rfsh0.clkf = clk1
rfsh0.setf = gnd
rfsh0.rstf = reset
rfsh0 = rfs0

; Probe 1

rfsh1.clkf = clk1
rfsh1.setf = gnd
rfsh1.rstf = reset
rfsh1 = rfsl1

; Data are read from the probe in blocks of 8 16 byte slices. An rfs pulse is generated every 64 clocks. cnt0 and cnt1 count the rfs pulses.

; Divide by 17 counter equations. If a falling edge of rfs has been detected, increment the counter. The counters are reset to 0 by a tfs/eop signal. rfs is divided by 17 because the dsp generates an rfs pulse at the start of each block read, and also at the end. This means 17 rfs pulses are generated for each block read.

; Probe 0

cnt0[4..0].clkf = clk1
cnt0[4..0].setf = gnd
cnt0[4..0].rstf = reset

if (/tfs0) then
begin
if (rfsh0 * /rfsl0) then
begin
  cnt0[2] = cnt0[2] :+: (cnt0[1] * cnt0[0])
  cnt0[1] = cnt0[1] :+: cnt0[0]
  cnt0[0] = /cnt0[0] * /cnt0[4]
end
else
begin
end
end
else
begin
end

36
cnt0[4..0] = cnt0[4..0]
end
else
begin
cnt0[4..0] = 0
end
end

;Probe 1
cntl[4..0].clkf = clk1
cntl[4..0].setf = gnd
cntl[4..0].rstf = reset

if (/tfs1) then
begin
if (rfsh1 * /rfsll) then
begin
cntl[0] = /cntl[0] * /cntl[4]
end
else
begin
  cntl[4..0] = cntl[4..0]
end
end
else
begin
  cntl[4..0] = 0
end
end

;Unload is generated from every other rfs pulse.
;The first unload of block read is not generated until the 2nd rfs.
;This prevents the slice loaded into the probe shift registers
;by the last unload of the previous block from being lost. It also
;allows the overload count to be read from the probe, prior to
;unloading the first slice of a particle.
;Unload is only asserted during the second half of the sclk cycle.
unld0 = rfs0 * /sclk0 * (cnt0[4] + cnt0[3] + cnt0[2] + cnt0[1]) *
       /cnt0[0]
       /cntl[0]

;Bitshift is enabled by the first rfs, but is delayed until the end of
;the rfs pulse. This avoids shifting away the first bit of data
;before it is read. Bitshift is disabled when the 9th rfs occurs at
;the end of a block read. During a tfs/eop bitshift is enabled and is
;the same as sclk. This is to shift the probe control word.
;Bitshift is the complement of sclk.
;Bitshift is inhibited during an unload.
           cnt0[0])+tfs0)
           cnt[0])+tfs1)
; FullO, fulll and chgirq are put onto the d0..d2 lines and are enabled
; by rdfull.
\[ d[2..0].\text{trst} = \text{rdfull} \]
\[ d[0] = \text{fullO} \]
\[ d[1] = \text{fulll} \]
\[ d[2] = \text{chgirq} \]

; Define the interrupt request sources edge detectors.
\[ \text{fullO} \text{cur}\.\text{clkf} = \text{clk1} \]
\[ \text{fullO} \text{cur}\.\text{setf} = \text{gnd} \]
\[ \text{fullO} \text{cur}\.\text{rstf} = \text{reset} \]
\[ \text{fullO} \text{cur} = \text{fullO} \]

\[ \text{fullO} \text{prev}\.\text{clkf} = \text{clk1} \]
\[ \text{fullO} \text{prev}\.\text{setf} = \text{gnd} \]
\[ \text{fullO} \text{prev}\.\text{rstf} = \text{reset} \]
\[ \text{fullO} \text{prev} = \text{fullO} \text{cur} \]

\[ \text{fullO} \text{int} = ((\text{fullO} \text{cur} * /\text{fullO} \text{prev}) + \text{fullO} \text{int}) * /\text{rdfull} \]

\[ \text{fulll} \text{cur}\.\text{clkf} = \text{clk1} \]
\[ \text{fulll} \text{cur}\.\text{setf} = \text{gnd} \]
\[ \text{fulll} \text{cur}\.\text{rstf} = \text{reset} \]
\[ \text{fulll} \text{cur} = \text{fulll} \]

\[ \text{fulll} \text{prev}\.\text{clkf} = \text{clk1} \]
\[ \text{fulll} \text{prev}\.\text{setf} = \text{gnd} \]
\[ \text{fulll} \text{prev}\.\text{rstf} = \text{reset} \]
\[ \text{fulll} \text{prev} = \text{fulll} \text{cur} \]

\[ \text{fulll} \text{int} = ((\text{fulll} \text{cur} * /\text{fulll} \text{prev}) + \text{fulll} \text{int}) * /\text{rdfull} \]

; Define the multiplexed DSP interrupt request output.
\[ \text{dirq2} = \text{fullO} \text{int} + \text{fullO} \text{int} + \text{chgirq} \]

; Load the TAS divider registers.
\[ \text{tdiv0}[1..0].\text{clkf} = \text{clk1} \]
\[ \text{tdiv0}[1..0].\text{setf} = \text{gnd} \]
\[ \text{tdiv0}[1..0].\text{rstf} = \text{reset} \]
if (prgtas0) then
begin
\[ \text{tdiv0}[1..0] = d[1..0] \]
end
else
begin
\[ \text{tdiv0}[1..0] = \text{tdiv0}[1..0] \]
end

\[ \text{tdiv1}[1..0].\text{clkf} = \text{clk1} \]
\[ \text{tdiv1}[1..0].\text{setf} = \text{gnd} \]
\[ \text{tdiv1}[1..0].\text{rstf} = \text{reset} \]
if (prgtas1) then
begin
\[ \text{tdiv1}[1..0] = d[1..0] \]
end
else
begin
  tdiv1[1..0] = tdiv1[1..0]
end

; TAS divider equations. TAS divisions of 2, 4, and 8 are available.
tcnt[3..0].clkf = tasbase
tcnt[3..0].setf = gnd
tcnt[3..0].rstf = reset
tcnt[0] = /tcnt[0]

; Select the TAS output based on the tas divider registers.
case (tdiv0[1..0])
begin
  0: begin tas0 = tcnt[0] end
  1: begin tas0 = tcnt[1] end
  2: begin tas0 = tcnt[2] end
  3: begin tas0 = tcnt[3] end
end

case (tdiv1[1..0])
begin
  0: begin tas1 = tcnt[0] end
  1: begin tas1 = tcnt[1] end
  2: begin tas1 = tcnt[2] end
  3: begin tas1 = tcnt[3] end
end

taspll = tcnt[1]
APPENDIX D

ADSP2101 Source Code Listings
/* vmegrey.h
   Defines for the VME Grey Scale Interface.
*/

#ifndef VMEGREY_H
#define VMEGREY_H
#define TRUE 1
#define FALSE 0

/* Note absolute addresses are 16 bit word addresses, not byte addresses. */

#define GREY_DPR_BASE 0x0000 /* dp ram dsp base addr */
#define GBL_CTL 0x0000 /* global ctrl struct addr */
#define PRB_0_CTL 0x0020 /* probe 0 ctrl struct addr */
#define PRB_1_CTL 0x0030 /* probe 1 ctrl struct addr */
#define GBL_BUF0 0x0080 /* probe0 global data buffer */
#define GBL_BUF1 0x1080 /* probe1 global data buffer */
#define DPR_SEM 0x2000 /* dp ram semaphore */
#define RDCCHG 0x3000 /* read charge data */
#define RDFULL 0x3001 /* read probe full bits */
#define LIRQ0 0x3002 /* interrupt request 0 */
#define LIRQ1 0x3003 /* interrupt request 1 */
#define SETTAS 0x3004 /* set true air speed */
#define PRGTAS0 0x3005 /* program tas 0 */
#define PRGTAS1 0x3006 /* program tas 1 */
#define PRGCHG 0x3007 /* program charge divider */

#define RDFULL_PRB0_MASK 0x01 /* probe 0 full bit mask */
#define RDFULL_PRB1_MASK 0x02 /* probe 2 full bit mask */
#define RDFULL_CHG_MASK 0x04 /* charge data avail bit mask */
#define PRGCHG_CNTS_PER_MM 20 /* charge divider counts per mm of charge res */

#define GREY_BUF_LEN 0x0F80 /* dual-port buf len (3968), (ints) */
#define GREY_BUF_FULL_IDX 2856 /* buf is full if index exceeds */
#define GREY_DPR_SIZE 8192 /* size of the dp memory (ints) */
#define GREY_EOP_DELAY 150 /* eop delay loop count */
#define GREY_MIN_SLICE 8 /* min num of slices per particle */
#define GREY_MAX_SLICE 128 /* max num of slices per particle */
#define GREY_NUM_PROBES 2 /* number of probes per interface */
#define GREY_BITS_PER_PIX 2 /* bits per pixel */
#define GREY_PIX_PER_WORD 8 /* pixels per 16 bit word */
#define GREY_PIX_PER_SLICE 64 /* pixels per 16 byte slice */
#define GREY_FIRST_PIX_PER_BIT 4 /* pixs per bit of pix mask*/
#define GREY_WORDS_PER_SLICE 8 /* 16 bit words per slice */
#define GREY_LEAST_PIX_MASK 0x0003 /* least significant pixel mask */
#define GREY_MOST_PIX_MASK 0x00FF /* most significant pixel mask */
#define SP_BUF_LEN 64 /* sport circ buf len (ints) */
#define CHG_BUF_LEN 128 /* charge data circ buf len (ints) */
#define CHG_NUM_SAMPS 64 /* total chg samps per part */
#define CHG_POS_MASK 0xFFFF /* set charge positive mask */
#define CHG_NEG_MAS 0xF000 /* set charge negative mask */
#define CHG_SIGN_MASK 0x0800 /* charge negative sign bit mask */
#define GREY_MAX_TAS 150 /* maximum tas (m/sec) */
```c
#define GREY_TAS_DACNT 4095 /* maximum tas d/a count */
#define G1 0x4731 /* ascii code for "G1" */
#define G2 0x4732 /* ascii code for "G2" */
#define GNONE 0
#define GA1 1
#define GA2 2
#define PROBE0 0
#define PROBE1 1
#define TCOUNT_MAX 49999 /* init tcount value, 100 msec ints*/
#define TSCALE_2US 24 /* init tscale value, 2 usec/count*/
#define TCOUNT_PER_MSEC 500 /* # counts per msec */
#define TCOUNT_SHFT_PER_USEC 1 /* #bits to shift to compute usec */

/* Probe resolution and tas divider definitions. */
#define RES_25 25
#define RES_50 50
#define RES_100 100
#define RES_200 200
#define TAS_DIV_25 0
#define TAS_DIV_50 1
#define TAS_DIV_100 2
#define TAS_DIV_200 3

/* Macros. */
#define prgtas0 *(int*)PRGTAS0
#define prgtasl *(int*)PRGTAS1
#define prgchg *(int*)PRGCHG
#define rdcharge *(int*)RDCHG
#define rdfull *(int*)RDFULL
#define settas *(int*)SETTAS
#define lirq(probe) *(int*)(LIRQO + probe) = 1
#define get_sem() for (*(int*)DPRSEM = 0; *(int*)DPRSEM & 0x01;
               *(int*)DPRSEM = 0)
#define rel_sem() *(int*)DPRSEM = 1

/* Swap words of a long int. Compiler does arithmetic left shifts. */
#define swap_words(x) (x << 16) + ((x >> 16) & 0x0000ffff)

/* Define the Probe and Global Control Structs. */
typedef struct { /* Probe control struct */
    short id;     /* probe id, 'G1', 'G2' */
    short id_fill; /* null bytes of the id string */
    short type;   /* 0 = none, 1 = GA1, 2 = GA2 */
    short res;    /* resolution(microns), 25 50 100 200 */
    short chg_spc; /* charge spacing (mm), 0 = none */
    short chg_loc; /* charge ring dist from beam (mm) */
    short control; /* control byte to send to the probe */
    short spare;
} GreyPctl;

typedef struct { /* global control struct */
    short full[GREY_NUM_PROBES]; /* buffer full lengths */
    short hour; /* set interface hour */
    short minute; /* current minute */
    short sec; /* current second */
```
short set_time;  /* set time flag */
short tas;  /* current true air speed (m/s) */
short go;  /* control structs initialized */
} GreyGctl;

/* Pms 2d grey scale particle structures. Each particle is recorded as a GreyParticle header, followed by the image data. The image data consists of slice_count GreySlice structures. */

typedef struct {
    short id;  /* probe id, 'G1', 'G2' */
    short id_fill;  /* null bytes of the id string */
    short hour;  /* particle time stamp, hour */
    short minute;  /* time stamp, minute */
    short sec;  /* time stamp, second */
    short msec;  /* time stamp, millisecond */
    long ena_sec;  /* probe enabled second count */
    long ena_usec;  /* probe enabled usec count */
    long acq_sec;  /* particle acquired second count */
    long acq_usec;  /* particle acquired usec count */
    short tas;  /* true air speed, m/sec */
    short width;  /* particle max width in pixels */
    short length;  /* particle max length in pixels */
    short probe_ctl;  /* probe control word */
    short ovld_cnt;  /* # particles while in overload */
    short slice_cnt;  /* # slices in the image data */
    short chg_len;  /* # 16-bit charge samples */
    short spare1;
    short spare2;
    short spare3;
} GreyParticle;

/* An image slice is defined by an array of GREY_BYTES_PER_SLICE bytes. */

typedef struct {
    unsigned short segment [GREY_WORDS_PER_SLICE];
} GreySlice;

#endif
/* vmegrey.c

ADSP2101 code for the VME Grey Scale interface.
NCAR/RAF
Jerry V. Pelk
*/

#include <adsp2101.h>
#include <signal.h>
#include <string.h>
#include "cdef2101.h"
#include "vmegrey.h"

/* Reserve registers for the serial port autobuffering. */
#pragma reserved 12

/* Program functions. */
void get_image_data (); /* get image data */
int process_image (); /* process image data */
int process_slice (); /* process an image slice */
int process_charge (); /* check charge sampling status */
void build_pix_mask (); /* build the pixel mask */
void init_particle (); /* initialize particle header */
void stamp_particle (); /* time stamp particle */
void send_eop (); /* send an eop to a probe */
void set_tas (); /* set true air speed */
void sport0_isr ();
void sport1_isr ();
void dirq2_isr ();
void timer_isr ();
void init_isrs (); /* initialize isrs */
extern void init_charge (int); /* init charge sampling */
extern void init_mem (); /* init system memory */
extern void init_sports (); /* init serial ports */
extern void init_tas (); /* initialize true air speed divider */
extern void check_time (); /* check for time set */
extern void enable_sport_rx (); /* enable sport to receive data */
extern int read_first_pix (); /* read first valid pixel */
extern void update_dual_port (); /* update dp buffer control vars */
extern void wait_vme (); /* wait for vme host initialization */

/* Circular data buffers. */
int circ sp_buf0[SP_BUF_LEN];
int circ sp_buf1[SP_BUF_LEN];
int circ chg_buf[GREY_NUM_PROBES][CHG_BUF_LEN];

/* Global variables. */
int gidx[GREY_NUM_PROBES]; /* current gbuf0 index */
int gstart[GREY_NUM_PROBES]; /* cur gbuf particle starting idx */
int sp_full[GREY_NUM_PROBES]; /* sport local buffer full flags */
int eop[GREY_NUM_PROBES]; /* end of particle flags */
int sport_enabled[GREY_NUM_PROBES]; /* serial port rx enabled flags */
GreySlice pix_mask[GREY_NUM_PROBES]; /* unused pixels mask */
int control; /* control word */
long tas; /* current tas, long to prevent ovf */
int *ptrl; /* needs to be global for inline */
/* Time variables. */
int new_sec; /* new second flag */
int hour; /* current hour */
int minute; /* current minute */
int sec; /* current second */
long usec; /* current microsecond */
long sec_cnt; /* seconds since startup */
long acq_sec[GREY_NUM_PROBES]; /* particle acquired second */
long acq_usec[GREY_NUM_PROBES]; /* particle acquired microsecond */
int acquired[GREY_NUM_PROBES]; /* particle acquired flag */

/* Dual-port memory variables. */
GreyGctl *gctl; /* control struct */
GreyPctl *pctl[GREY_NUM_PROBES]; /* probe control structs */
GreyParticle *part[GREY_NUM_PROBES]; /* current probe particle headers */
int *gbuf[GREY_NUM_PROBES]; /* data buffers */

/* Charge sampling buffers and control variables. */
int chg_start[GREY_NUM_PROBES]; /* charge starting index */
int chg_end[GREY_NUM_PROBES]; /* charge ending index */
int chg_idx[GREY_NUM_PROBES]; /* current charge sample index */
int chg_offset[GREY_NUM_PROBES]; /* charge offset delay index */
int chg_full[GREY.NUM_PROBES]; /* charge buffer full flag */

main () {
    static int full;
    static int probe;

    init_mem (); /* init system memory */
    probe = PROBE0;

    /* Wait for the vme host to initialize the control structs. */
    wait_vme();

    init_sports ();
    init_isrs ();

    /* Initialize the probes. */
    init_tas (PROBE0);
    init_tas (PROBE1);
    init_charge(PROBE0);
    init_charge(PROBE1);
    settas = 500;

    while (TRUE) { /* main loop */
        if (new_sec) {
            set_tas (); /* set tas once per second */
            check_time(); /* check for time set */
            new_sec = FALSE;
        }
    }
/* If the dual-port buffer is not full, and data is available from a
probe, then process the data. */
if (pctl[probe]->type != GNONE) {
    /* Get the buffer full flag. */
    get_sem();
    full = gctl->full[probe];
    rel_sem();

    /* If the dual port buffer is full, don't do anything. */
    if (full) {
        if (eop[probe]) {
            init_particle (probe);
            send_eop (probe);
            eop[probe] = FALSE;
        }
    }

    /* If image data is available from the probe, go read the data. */
    if (((probe == PROBEO) && (rdfull & RDFULL_PRBO_MASK)) ||
        ((probe == PROBE1) && (rdfull & RDFULL_PRB1_MASK))) {
        get_image_data (probe);
    }

    /* Process the image data. If an eop is detected, wait for the charge
data. */
    if (eop[probe] = process_image (probe)) {
        while (!process_charge (probe));
        update_dual_port (probe);
    }
    spfull[probe] = FALSE;
}

/* Toggle probes. */
if (probe == PROBEO)
    probe = PROBE1;
else
    probe = PROBEO;

}*/

void get_image_data (int probe)
/* Gets an image data block from a probe. */
{
    int temp[4];

    enable_sport_rx (probe);            /* enable the sport */
#pragma inline
    dm(ptr1_) = I2;
    dm(ptr2_) = I2;
#pragma inline

    /* Occasionally the sport isr cannot shut the sport off before the
first few words are overwritten. Once the first 4 words have been received, they are saved, and then restored when the read completes.

while ((int)ptr2 < (int)(ptr1 + 4)) { /* wait for first 4 words */
    #pragma inline
    dm(ptr2_) = 12;
    #pragma inline

    temp[0] = *ptr1;                      /* save the first 4 words */
    temp[1] = *(ptr1 + 1);
    temp[2] = *(ptr1 + 2);
    temp[3] = *(ptr1 + 3);

    while (!sp_full[probe]);          /* wait for completion */

    *ptr1 = temp[0];                         /* restore first 4 words */
    *(ptr1 + 1) = temp[1];
    *(ptr1 + 2) = temp[2];
    *(ptr1 + 3) = temp[3];
}

int process_image (int probe)

/* Processes image data blocks read from a probe. */
{
    int start_idx;                      /* local buffer starting index */
    int sp_idx;                        /* local buffer current index */
    int null_slice;                    /* end of particle flag */
    int first_pix;
    int *sp_buf;

    if (probe == PROBEO)
        sp_buf = sp_buf0;
    else
        sp_buf = sp_buf1;

    /* If this is the start of a new particle, compute the slice mask and fill in the particle header. */
    if (!part[probe]->slice_cnt) {
        stamp_particle (probe, *(sp_buf + 1));
        first_pix = read_first_pix (probe, *sp_buf);
        build_pix_mask (probe, first_pix);
        start_idx = GREY_WORDS_PER_SLICE;
    } else
        start_idx = 0;

    for (sp_idx = start_idx, null_slice = FALSE;
         (sp_idx < SP_BUF_LEN) && !null_slice;
         sp_idx += GREY_WORDS_PER_SLICE) {
        part[probe]->slice_cnt++;
        if (!process_slice (probe, sp_buf + sp_idx) &&
int process_slice (int probe, int *src)
/* Processes a slice of data. Return FALSE if the slice is empty. */
{
    int j;
    int k;
    int t;
    int first_pix = -1;
    int last_pix = -1;
    int pix_cnt = 0;

    first_pix = 0;
    last_pix = -1;
    pix_cnt = 0;

    for (j = 0; j < GREY_WORDS_PER_SLICE; j++) {
        /* The GA1 probe data is positive true logic, convert to negative true logic to be compatible with the other 2d probes. The GA1 also only shifts out the minimum number of needed bits per slice. Apply the pixel mask to convert the unshifted bits to ones. */
        if (pctl[probe]->type == GA1)
            *(src + j) = -* (src + j) | pix_mask[probe].segment[j];

        /* Compute the particle width in this slice. */
        if (t = -* (src + j)) {
            for (k = 0; k < GREY_PIX_PER_WORD; k++, t<<= GREY_BITS_PER_PIX) {
                pix_cnt++;
                if (t & GREY_MOST_PIX_MASK) {
                    if (!first_pix)
                        first_pix = pix_cnt;
                    last_pix = pix_cnt;
                }
            }
        }
    }

    /* Update the particle header statistics. */
    if (first_pix) {
        part[probe]->length++;
        if ((k = last_pix - first_pix + 1) > part[probe]->width)
            part[probe]->width = k;
    }
    else
        gidx[probe] += sp_idx - start_idx;
}
}
int process_charge (int probe)

/* Checks for charge data collection completion. Moves charge data to
the dual-port buffer. Returns TRUE if the charge data is
completed, FALSE otherwise. */
{
  int j;
  int k;

  /* If charge sampling is not enabled, just return TRUE. */
  if (lchg_offset[probe])
    return TRUE;

  /* If charge sampling has completed, move the charge data to the dp
  buffer. */
  if (chg_full[probe]) {
    for (k = 0, j = chg_start[probe]; k < CHG_NUM_SAMPS; k++, j++) {
      if (j >= CHG_BUF_LEN)
        j = 0;
      /* If the charge is negative, set the upper 4 bits, otherwise clear
      them. */
      if (chg_buf[probe][j] & CHG_SIGN_MASK)
      else
    }
    part[probe]->chg_len = CHG_NUM_SAMPS;
  } else
    return FALSE; /* still waiting for charge data */

  return TRUE;
}

void init_particle (int probe)

/* Initializes a particle’s header. */
{
  int j;
  long cur_usec;

  memset ((void*)part[probe], 0, (size_t)sizeof(GreyParticle));

  /* Stamp the probe enable time. */
  cur_usec = usec + ((TCOUNT_MAX - Tcount_Reg) << TCOUNT_SHFT_PER_USEC);
part[probe]->ena_usec = swap_words(cur_usec); /* swap words of long */
part[probe]->ena_sec = swap_words(sec_cnt);
acquired[probe] = FALSE;
}
/**********************************************************************/
void stamp_particle (int probe, int ovld_cnt)
/* Stamps the particle header information. */
{
part[probe]->id = pctl[probe]->id;
part[probe]->tas = tas;
part[probe]->probe_ctl = pctl[probe]->control;
part[probe]->acq_usec = swap_words(acq_usec[probe]); /* swap long */
part[probe]->acq_sec = swap_words(acq_sec[probe]);
part[probe]->hour = hour;
part[probe]->minute = minute;
part[probe]->sec = sec;
part[probe]->msec = usec / 1000;
/* Overload count in upper 8 bits. Compiler does arithmetic left
shifts. */
part[probe]->ovld_cnt = (~ovld_cnt >> 8) & 0x00ff;
}
/**********************************************************************/
void build_pix_mask (int probe, int first_pix)
{
int mask_idx;
int first_idx;
int bit_idx;

for (mask_idx = 0; mask_idx < sizeof (GreySlice); mask_idx++)
pix_mask[probe].segment[mask_idx] = 0;
for (mask_idx = GREY_WORDS_PER_SLICE - 1, first_idx = 0;
(mask_idx >= 0) && (first_idx < first_pix); mask_idx--)
{
for (bit_idx = 0; (bit_idx < GREY_PIX_PERWORD) &&
(first_idx < first_pix); bit_idx++, first_idx++)
{
pix_mask[probe].segment[mask_idx] <<= GREYBITSPERPIX;
pix_mask[probe].segment[mask_idx] |= GREY_LEAST_PIX_MASK;
}
}
/**********************************************************************/
void send_eop (int probe)
{
int j;
get_sem();
control = pctl[probe]->control;
rel_sem();
if (probe == PROBE0) {

52
Sport0_Ctrl_Reg |= ISCLK; /* enable internal serial clock */
Sys_Ctrl_Reg |= SPORT0_EN; /* enable the sport */
#pragma inline
TX0 = dm(control_);
#pragma inline
for (j = 0; j < 150; j++); /* wait for control word to be sent */
Sport0_Ctrl_Reg &= ~ISCLK; /* shut off the serial clock */
Sys_Ctrl_Reg &= ~SPORT0_EN; /* disable the sport */
}

else {
    Sport1_Ctrl_Reg |= ISCLK; /* enable internal serial clock */
    Sys_Ctrl_Reg |= SPORT1_EN; /* enable the sport */
#pragma inline
    TX1 = dm(control_);
#pragma inline
    for (j = 0; j < 150; j++); /* wait for control word to be sent */
    Sport1_Ctrl_Reg &= ~ISCLK; /* shut off the serial clock */
    Sys_Ctrl_Reg &= ~SPORT1_EN; /* disable the sport */
}

sport_enabled[probe] = FALSE;
}

void set_tas ()
/* Writes the true air speed count to the tas generator. */
{
    get_sem();
tas = (long)gctl->tas;
rel_sem();

    settas = (int)((tas * GREY_TAS_DACNT) / GREY_MAX_TAS);
}

void sport0_isr ()
/* Sport 0 interrupt service routine. Called when a buffer has been
filled. */
{
    Sport0_Ctrl_Reg &= ~(IRFS + ISCLK); /* disable intern rfs and ser clk*/
    Sys_Ctrl_Reg &= ~SPORT0_EN; /* disable the sport */
    Sport0_Autobuf_Ctrl = 0x00; /* disable receive autobuf */
    sp_full[PROBE0] = TRUE; /* set sport buffer full flag */
    sport_enabled[PROBE0] = FALSE;
}

void sport1_isr ()
/* Sport 1 interrupt service routine. Called when a buffer has been
filled. */
{
    Sport1_Ctrl_Reg &= ~(IRFS + ISCLK); /* disable intern rfs and ser clk*/
void dirq2_isr ()
/* External interrupt 2 service routine. Called when a probe first
becomes full or when charge data is available. */
{
  int rd_full;
  int cur_cnt;
  int j;

  /* Read the probe and charge full status word. */
  rd_full = rdfull;

  /* Read the current timer count. */
  cur_cnt = Tcount_Reg;

  /* Read the charge data */
  if (rd_full & RDFULL_CHG_MASK) {
    for (j = 0; j < GREY_NUM_PROBES; j++) {
      chg_buf[j][chg_idx[j]] = rdcharge;
      chg_buf[j][chg_idx[j]] = rdcharge;
    }
  }

  /* Check for chg_idx wraparound and for charge completion. */
  if (chg_idx[j] >= CHG_BUF_LEN)
    chg_idx[j] = 0;
  if (chg_idx[j] == chg_end[j])
    chg_full[j] = TRUE;
  }

  /* If this is the start of a new particle, stamp the acquisition
  counters. */
  if (((rd_full & RDFULL_PRB0_MASK) & lacquired[PROBE0]) { /* prb 0 */
    acq_usec[PROBE0] = usec + ((TCOUNT_MAX - cur_cnt) <<
                              TCOUNT_SHFT_PER_USEC);
    acq_sec[PROBE0] = sec_cnt;
    acquired[PROBE0] = TRUE;

    /* Calculate the charge buffer start index from the current sample
    index. */
    if ( (chg_start[PROBE0] = chg_idx[PROBE0] - chg_offset[PROBE0])< 0)
      chg_start[PROBE0] += CHG_BUF_LEN;
    if ( (chg_end[PROBE0] = chg_start[PROBE0] + CHG_NUM_SAMPS) >=
        CHG_BUF_LEN)
      chg_end[PROBE0] -= CHG_BUF_LEN;
    chg_full[PROBE0] = FALSE;
  }
}
if ((rd_full & RDFULL_PRB1_MASK) && !acquired[PROBE1]) { /* prb 1 */
    acq_usec[PROBE1] = usec + ((TCOUNT_MAX - cur_cnt) << TCOUNT_SHFT_PER_USEC);
    acq_sec[PROBE1] = sec_cnt;
    acquired[PROBE1] = TRUE;

    /* Calculate the charge buffer start index from the current sample index. */
    if ((chg_start[PROBE1] = chg_idx[PROBE1] - chg_offset[PROBE1]) < 0)
        chg_start[PROBE1] += CHG_BUF_LEN;
    if ((chg_end[PROBE1] = chg_start[PROBE1] + CHG_NUM_SAMPS) >= CHG_BUF_LEN)
        chg_end[PROBE1] -= CHG_BUF_LEN;
    chg_full[PROBE1] = FALSE;
}

void timer_isr ()
/* Timer interrupt service routine. Called every 100 msec. */
{
    if ((usec += (long)100000) >= (long)1000000) {
        new_sec = TRUE;
        sec_cnt++;
        usec = 0;
        if (++sec >= 60) {
            sec = 0;
            if (++minute >= 60) {
                minute = 0;
                if (++hour >= 24) {
                    hour = 0;
                }
            }
        }
    }
}

void init_isr ()
/* Initializes the interrupt service routines, and enables interrupts. */
{
    /* Attach isrs. */
    interrupt (SIGSPORTORECV, sport0_isr);
    interrupt (SIGINT0, sport1_isr);
    interrupt (SIGTIMER, timer_isr);
    interrupt (SIGINT2, dirq2_isr);

    /* Program the timer counter to decrement every 2 usec (25 clocks),
    and interrupt every 100 msec. */
    Tscale_Reg = TSCALE_2US;
    Tperiod_Reg = TCOUNT_MAX;
    Tcount_Reg = TCOUNT_MAX;

    /* Calculate the charge buffer start index from the current sample index. */
    if ((chg_start[PROBE1] = chg_idx[PROBE1] - chg_offset[PROBE1]) < 0)
        chg_start[PROBE1] += CHG_BUF_LEN;
    if ((chg_end[PROBE1] = chg_start[PROBE1] + CHG_NUM_SAMPS) >= CHG_BUF_LEN)
        chg_end[PROBE1] -= CHG_BUF_LEN;
    chg_full[PROBE1] = FALSE;
}
#pragma inline
/* interrupt nesting, edge sensitivity for irq2 */
ICNTL = H#00;
/* enable sport0 & sport1 recv, timer, and irq2 ints */
IMASK = H#2B;
ENA TIMER;
#pragma inline
}
/* interrupt nesting, edge sensitivity for irq2 */
ICNTL = H#00;
/* enable sport0 & sport1 recv, timer, and irq2 ints */
IMASK = H#2B;
ENA TIMER;
#pragma inline
}
/*********************/
/* extpm.c

ADSP2101 module contains routines for the vme grey scale interface
which are only run on startup, or which execute infrequently.*/

#include <string.h>
#include "cdef2101.h"
#include "vmegrey.h"

void init_charge (int);  /* initialize charge sampling */
void init_mem (); /* init system memory */
void init_sports (); /* init serial ports */
void init_tas (); /* initalize treu air speed divider */
void check_time (); /* check for time set */
void enable_sport_rx (); /* enable sport to receive data */
int read_first_pix (); /* read first valid pixel */
void update_dualport (); /* update dp buffer control vars */
void wait_vme (); /* wait for vme host initialization */

extern int circ sp_buf0[SP_BUF_LEN];
extern int circ sp_buf1[SP_BUF_LEN];
extern int gidx[GREY_NUM_PROBES]; /* current gbuf0 index */
extern int gstart[GREY_NUM_PROBES]; /* cur part start idx */
extern int sp_full[GREY_NUM_PROBES]; /* sport local buf full flags */
extern int eop[GREY_NUM_PROBES]; /* end of particle flags */
extern int sport_enabled[GREY_NUM_PROBES]; /* serial port rx enab */

/* Dual-port memory variables. */
extern GreyGctl *gctl;
extern GreyPctl *pctl[GREY_NUM_PROBES]; /* probe control structs */
extern GreyParticle *part[GREY_NUM_PROBES]; /* cur probe part hdrs*/
extern int *gbuf[GREY_NUM_PROBES]; /* data buffers */

/* Time variables. */
extern int new_sec; /* new second flag */
extern int hour; /* current hour */
extern int minute; /* current minute */
extern int sec; /* current second */
extern long usec; /* current microsecond */
extern long sec_cnt; /* seconds since startup */
extern int acquired[GREY_NUM_PROBES]; /* particle acquired flag */

/* Charge sampling buffers and control variables. */
extern int circ chg_buf[GREY_NUM_PROBES][CHG_BUF_LEN];
extern int chg_start[GREY_NUM_PROBES]; /* charge starting index */
extern int chg_end[GREY_NUM_PROBES]; /* charge ending index */
extern int chg_idx[GREY_NUM_PROBES]; /* current charge index */
extern int chg_offset[GREY_NUM_PROBES]; /* charge offset index */
extern int chg_full[GREY_NUM_PROBES]; /* charge buf full flag */

void init_charge (int probe)
/ Initialize the charge sampling for a probe. */
{
    if (Ipctl[probe]->chg_spc)
        return;

    /* Compute the charge tas divider value. The tasbase signal is a 0-12
     MHz signal corresponding to a tas of 0-150 m/s. The charge tas
deriver is a 10 bit down counter. The upper 8 bits are
programmable. tasbase is divided by 4 * the prgchg count programmed
into the charge divider here. The charge sample timing is the same
for both probes. */

    prgchg = pctl[probe]->chg_spc * PRGCHG_CNTS_PER_MM;

    /* Compute the charge buffer index offset based upon the distance from
the probe laser beam to the center of the charge ring, and the
charge sample resolution. The multiplication by 2 is to account
for the index offset for both the charge and splash samples. */

    chg_offset[probe] = (pctl[probe]->chg_loc * 2) /
                        pctl[probe]->chg_spc;
}

/***************************************************************************/

void init_mem ()

    /* Initialize the system memory. */
{
    int j;

    Dm_Wait_Reg = DWAIT0_1 + DWAIT1_1 + DWAIT2_1 + DWAIT3_1;
    Sys_Ctr1_Reg &= PWAIT_MAK; /* zero wait states for ext pm */
    rdfull = 3;

    /* Zero the dual-port memory. */
    get_sem();
    memset ((void*)GREY_DPR_BASE, 0, (size_t)GREY_DPR_SIZE);
    rel_sem();

    gctl = (GreyGctl*)GBL_CTL;
    pctl[0] = (GreyPctl*)PRB_0_CTL;
    pctl[1] = (GreyPctl*)PRB_1_CTL;
    gbuf[0] = (int*)GBL_BUF0;
    gbuf[1] = (int*)GBL_BUF1;

    for (j = 0; j < GREY_NUM_PROBES; j++) {
        part[j] = (GreyParticle*)gbuf[j];
        gidx[j] = sizeof(GreyParticle);
        gstart[j] = 0;
        sp_full[j] = FALSE;
        sport_enabled[j] = FALSE;
        eop[j] = TRUE; /* send an initial eop */
        chg_start[j] = 0;
        chg_end[j] = 0;
        chg_idx[j] = 0;
        chg_offset[j] = 0;
}
chg_full[j] = FALSE;
acquired[j] = FALSE;
}
new_sec = FALSE; /* zero time variables */
hour = 0;
minute = 0;
sec = 0;
usec = 0;
sec_cnt = 0;
}/**'/

void init_sports(){
/* Initialize the serial ports. */
{
/* Program the serial clock rate for approximately 2 Mhz. */
SportO_Sclkdiv = 2;
Sportl_Sclkdiv = 2;
/* Require tfs and no rfs,
normal receive framing mode, alternate transmit framing mode,
generate rfs and tfs internally
16 bit word length.
*/
SportO_Ctrl_Reg = TFSR + TFSW + ITFS + SLEN_16;
Sportl_Ctrl_Reg = TFSR + TFSW + ITFS + SLEN_16;
/* Program rfs to be issued every 64 bits. */
SportO_Rfsdiv = 63;
Sportl_Rfsdiv = 63;
Sys_Ctrl_Reg |= SPORT1_CFG; /* config sportl 1 as serial port */
}/**'/

void check_time(){
/* Checks for a time set. */
{
get_sem();
if (gctl->set_time) {
    hour = gctl->hour;
    minute = gctl->minute;
    sec = gctl->sec;
    gctl->set_time = FALSE;
}
rel_sem();
}/**'/

void init_tas(int probe){
/* Program the tas divider registers. */
{
59
int *progtas;

if (probe == PROBEO)
    progtas = (int*)PRGTAS0;
else
    progtas = (int*)PRGTAS1;

if ((pctl[probe]->res == RES_50)
    *progtas = TAS_DIV_50;
else if (pctl[probe]->res == RES_100)
    *progtas = TAS_DIV_100;
else if (pctl[probe]->res == RES_200)
    *progtas = TAS_DIV_200;
else
    *progtas = TAS_DIV_25;
}

void enable_sport_rx (int probe)

/* Enables sport to read a block of 8 sixteen byte image slices. */
{
    if (probe == PROBEO) {
        #pragma inline /* Set up autobuffering. */
        I2 = ^sp_buf0_;  
        M1 = 1;
        L2 = %spbuf0_;  
        #pragma inline 
        Sport0_Autobuf_Ctrl = 0x25; /* enable receive 
        autobuffering */
        Sport0_Ctrl_Reg |= IRFS + ISCLK; /* enab intern rfs and ser clk*/
        Sys_Ctrl_Reg |= SPORT0_EN; /* enable the sport */

    }
    else {
        #pragma inline /* Set up autobuffering. */
        I2 = ^sp_buf1_;  
        M1 = 1;
        L2 = %spbuf1_;  
        #pragma inline 
        Sport1_Autobuf_Ctrl = 0x25; /* enable receive 
        autobuffering */
        Sport1_Ctrl_Reg |= IRFS + ISCLK; /* enab intern rfs and ser clk*/
        Sys_Ctrl_Reg |= SPORT1_EN; /* enable the sport */

    }

    sport_enabled[probe] = TRUE;
}

int read_first_pix (int probe, int first_pix)

/* Calculates the first valid pixel from the first pixel bit pattern. 
   This value is only valid for the GA1 probes. The GA2 probes do not
mask unused pixels.
*/
{
    int j;
    if (pctl[probe]->type != GA1)
        return 0;
    for (j = 0; (j < GREY_PIX_PER_SLICE) && (first_pix & 0x8000);
        j += GREY_FIRST_PIX_PER_BIT, first_pix <<= 1);
    return j;
}  
*******************************************************************************/

void update_dual_port (int probe)
/* Checks if the dual-port buffer for the specified probe is full.
   Updates particle pointer and index. */
{
    /* If the dp buffer is full, mark the buffer full by writing the
       length into the global control struct, and reset the dp buffer
       index. */
    if (gidx[probe] > GREY_BUF_FULL_IDX) {
        get_sem();
        gctl->full[probe] = gidx[probe];
        rel_sem();
        gidx[probe] = 0;
        lirq(probe);  /* generate vme interrupt */
    }
    /* Update the particle pointer and buffer index. */
    part[probe] = (GreyParticle*)(gbuf[probe] + gidx[probe]);
    gidx[probe] += sizeof (GreyParticle);
}  
*******************************************************************************/

void wait_vme ()
/* Wait for the host vme processor to init the control structs. */
{
    int j;
    static int start;
    start = FALSE;
    while (!start) {
        get_sem();
        start = gctl->go;
        rel_sem();
        for (j = 0; j < 100; j++);
    }
}  
*******************************************************************************/