System Operation

5. SYSTEM OPERATION

5.1 System Control and Initialization

The unique aspects of system initialization for the CL-GD546X are discussed in this section. In particular, the programmer needs to be aware of the requirements of Rambus memory system initialization, and initialization of the 2D graphics accelerator and frame buffer memory. Example software, developed for use in the BIOS and display driver modules is presented as an aid to the programmer.

5.1.1 Rambus® Initialization

The programmer must initialize the Rambus RDRAMs before using them. The initialization process entails initializing the RIF Control, the RAC Control, and some of the Rambus RDRAM device registers. The Device Id register is the one of most interest. The Device Id register allows the programmer to assign a unique address that determines the address range, which it decodes. The Rambus RDRAM initialization procedure is listed below.

1) Reset the Rambus using the RAC Control register EXTBE.

2) Initialize the RIF Control register.

3) Broadcast the default values for Mode, DeviceId, and Delay.

4) Set the RIF initialization bit, due to delay timing changes.

5) Make changes for each device; assign a unique Id, enable the device <Mode Register Device Enable>, check that the device is enabled < if not exit loop>, and read the DeviceType determining its size.

6) Set the RIF Control register, enabling refresh and the number of banks (one of which was determined in Step 5).

7) For each DRAM, perform the current calibration algorithm.

NOTE: Memory-Mapped registers, Rambus transaction, and Rambus data must perform bus arbitration before a write cycle can be completed. Due to this fact, the NACK bit is set in R/F register prior to performing a transaction. The code must wait for the Ack bit pattern to determine when the write cycle has completed. This must be performed when the STOP_REFRESH (RIF) bit is set.

The following sample code illustrates how to initialize the Rambus.

```
;*************************************************************
;*************************************************************
;*
;*         ********************************************
;*         * Copyright (c) 1994-1996, Cirrus Logic, Inc. *
;*         *            All Rights Reserved             *
;*         ********************************************
;*
;* PROJECT: Laguna (CL-GD546X)
;*
;* FILE: rambus.asm
;*
;* DESCRIPTION: RAMBUS interface routines for Laguna BIOS.
```

```
;*
;* $Log: $
;*
;*****************************************************************
;*****************************************************************

;*------------------------------ MACROS ----------------------*;

;*****************************************************************
;*
;* MACRO NAME: DELAY
;*
;* DESCRIPTION: Burns some CPU Time
;*
;* RETURNS: al = Unknown
;*
;* REVISION HISTORY:
;*
;*****************************************************************
DELAY         MACRO
              call        RamBus_Dlay
ENDM
;*****************************************************************
;*
;* MACRO NAME: RACK
;*
;* DESCRIPTION: Wait for Ack
;*
;* RETURNS:
;*
;* REVISION HISTORY:
;*
;*****************************************************************
RACK          MACRO
              call        RamBus_Ack
ENDM
;*****************************************************************
;*
;* MACRO NAME: RNACK
;*
;* DESCRIPTION: Set RAC_CTL to NACK
;*
```

```
;* RETURNS: al = Unknown
;*
;* REVISION HISTORY:
;*
;*****************************************************************
RNACK         MACRO
              MEMOR16     RIF_CTL, RIF_NACK       ; Set Nack Bits
ENDM
;*****************************************************************
;*
;* MACRO NAME: DELAY
;*
;* DESCRIPTION: Burns some CPU Time
;*
;* RETURNS: al = Unknown
;*
;* REVISION HISTORY:
;*
;*****************************************************************

              .386
              include memmap.inc
              include rambus.inc
              include config.inc
              include options.inc
              .386

cgroup        group       VGA_Segment
              VGA_Segment SEGMENT PUBLIC BYTE use16
              PUBLIC      RamBus_Init
              ASSUME      CS:cgroup
              EXTRN       GetMemMap:NEAR
              EXTRN       getreg:NEAR
              EXTRN       Gate_Timer:NEAR
              ASSUME      DS:Memory_Mapped_Register
              ASSUME      ES:Memory_Mapped_Register
RAMBUS_DEFAULT:
              dw          8003h       ; Broadcast to Mode
              dd          DEFMODEREGVAL OR MODE_CE ; Default Mode
;             dw          8006h       ; Broadcast to RasInterval
;             dd          DEFRASREGVAL; Default Value
              dw          8001h       ; Broadcast to Device ID
```



```
                dd          003Ch       ; Default Device ID to Max
                dw          8002h       ; Broadcast to Delay
                dd          DEFDELAYREGVAL; Default Delay
RAMBUS_CNT = ($ - RAMBUS_DEFAULT) / 6
;
; This Table is used to encode current calibration
;
ENCODE_TABLE:
                db          C0SHIFT, C0WMASK ; Mask and Shift Value when a word
                db          C1SHIFT, C1WMASK ; Mask and Shift Value when a word
                db          C2SHIFT, C2WMASK ; Mask and Shift Value when a word
                db          C3SHIFT, C3WMASK ; Mask and Shift Value when a word
                db          C4SHIFT, C4WMASK ; Mask and Shift Value when a word
                db          C5SHIFT, C5WMASK ; Mask and Shift Value when a word
ENCODE_CNT = ($ - ENCODE_TABLE) SHR 1; Number of Words
;
; This Table is used to encode current calibration
;
DECODE_TABLE:
                dd          C0RMASK ; Mask Value
                db          C0SHIFT ; Shift Value
                dd          C1RMASK ; Mask Value
                db          C1SHIFT ; Shift Value
                dd          C2RMASK ; Mask Value
                db          C2SHIFT ; Shift Value
                dd          C3RMASK ; Mask Value
                db          C3SHIFT ; Shift Value
                dd          C4RMASK ; Mask Value
                db          C4SHIFT ; Shift Value
                dd          C5RMASK ; Mask Value
                db          C5SHIFT ; Shift Value
DECODE_CNT = ($ - DECODE_TABLE) / 5; Number of 5 Bytes

s               equ         short
;---------------------------------------------------------------
;
;               RamBus_Init()
;
;               Function:
;                  Called from POST to initialize the RamBus and compute size
;               of memory
;
```

```
;             Input:
;                None
;             Returns:
;                ax = Memory Size
;             Destroys
;             EAX, EBX, ECX, EDX, DI, ESI
;-----------------------------------------------------------------
RamBus_Init PROC          NEAR
                          push        ds
                          push        es

                    call          GetMemMap             ; Get Segment of Memory
                                                        ; Mapped Registers
                    mov           es, ax
                    mov           ds, ax

                    ;
                    ; Fix the Clock Register
                    ;
                    MEMWR16 TEST_REG, 01000h            ; Fix the Test Register
                    DELAY
                    MEMWR16 RIF_CTL, RIF_DEFAULT        ; Init RIF
                    DELAY

                    MEMWR16 RAC_CTL, 0485fh             ; Reset RamBus
                    DELAY
                    MEMWR16 RAC_CTL, 081fh              ; Clear it
                    DELAY

                    mov           ax, cs                ; Get Code Segment
                    mov           ds, ax
                    ASSUME  DS:cgroup
                    mov           cx, RAMBUS_CNT        ; RamBus Count
                    mov           si, OFFSET RAMBUS_DEFAULT

RAM_LOOP:
                    lodsw                               ; Get Register
                    MEMWR16 RAMBUS_TRANS, ax
                    RNACK
                    lodsd                               ; Get Value in eax
                    MEMWR32 RAMBUS_DATA, eax
                    RACK
```



```
                dec         cx                  ; More
                jnz         RAM_LOOP            ;

                mov         ax, es              ; Get Memory-Mapped
Segment
                mov         ds, ax
                ASSUME  DS:Memory_Mapped_Register
                MEMOR16 RIF_CTL, RIF_INIT       ; Init after Delay

                ;
                ; Assume they are all 16M Parts
                ;

                xor         ebx, ebx            ; 16M Counter
                mov         esi, 020h           ; 8M Start on 08M
Boundary
                mov         ecx, 028h           ; 8M Start on 0AM
Boundary
                xor         al, al              ; Start at 0
                xor         edx, edx            ; Ditto
RAM_16M:
                add         al, dh              ; Next Address
                mov         dh, 0Fh             ; Address Next
                mov         dl, REG_DEVID       ; Device Id
                xchg        eax, ebx            ; Write Value for Write
                call        RamBus_Write
                xchg        eax, ebx            ; Restore
                mov         dh, al              ; Restore Address
                mov         dl, REG_MODE        ; Mode
                mov         eax, DEFMODEREGVAL OR MODE_CE OR MODE_DE
                call        RamBus_Write
                RNACK
                MEMRD32     eax, RAMBUS_DATA ; Get it
                RACK
                and         al, MODE_DE         ; Is it set
                jz          s CHK_8BIT          ; No All Done

                ;
                ; Do the Device Manufacturer
                ;
                mov         dl, REG_DMAN        ; Device Manufacturer
                call        RamBus_Read
                push        edx                 ; Save dx
```

```
                mov         edx, eax            ; Save Read Value
                mov         eax, DEFRASTOSVAL; Assume Toshiba
                cmp         edx, TOSHIBA_ID     ; I was here
                pop         edx                 ; Restore dx
                je          s TOS_RAS           ; Are You??
                mov         eax, DEFRASNECVAL   ; Must be NEC
TOS_RAS:
                mov         dl, REG_RASINTERVAL ; Set RasInterval
                call        RamBus_Write

                ;
                ; Figure Out Size
                ;
                mov         dl, REG_DEVTYPE     ; Device Type
                call        RamBus_Read
                test        ax, SIZE_16M        ; 16 MB Part
                jz          s ITSA_8M           ; 8 M Part?
                test        ax, PARITY_BIT      ; Parity Bit
                jz          s ITSA_8BIT         ; 8 Bits Per Pixel
                add         bx, INC_16M         ; Next 16MB
                mov         al, 2h              ; Next Address
                or          edx, 10000h         ; Set 9-Bit DRAM Flag
                jmp         RAM_16M

                ;
                ; Not Sure where to put it so put it 8MB and below
                ; ITSA_8BIT:
                mov         dl, REG_DEVID       ; Put in Device ID
                xchg        eax, esi
                call        RamBus_Write
                xchg        eax, esi
                sub         esi, INC_16M        ; Next Address
                xor         al, al              ; Same Address
                jmp         RAM_16M

                ;
                ; Not Sure where to put it so put it above 0AM
                ; ITSA_8M:
                mov         dl, REG_DEVID       ; Put in Device ID
                xchg        eax, ecx
                call        RamBus_Write
                xchg        eax, ecx
```



```
                add         cx, INC_8M          ; Next Address
                xor         al, al              ; Same Address
                jmp         RAM_16M
                ; CHK_8BIT:
                mov         ax, si
                sub         ax, 20h             ; Clear Offset
                or          ax, ax              ; Any 2M w/ 8 Bit ?
                jz          s SKIP_8BIT         ; No All Done

                ;
                ; Fix Up 16M 8-Bit Parts
                ;
                neg         ax                  ; Make Positive
                shr         ax, 3               ; Fix Counter
                mov         dx, 801h            ; Address > 8M
RAM_8BIT:
                xchg        eax, ebx
                call        RamBus_Write
                xchg        eax, ebx
                add         bx, INC_16M         ; Next Address
                sub         dh, 2               ; Next 16M
                dec         ax                  ; More
                jnz         s RAM_8BIT

SKIP_8BIT:
                sub         cx, 028h            ; Clear Offset
                or          cx, cx              ; Any 8M ?
                jz          s SKIP_8M           ; No All Done

                ;
                ; Fix Up 8M Parts
                ;
                shr         cx, 2               ; Fix Counter
                mov         dx, 0A01h           ; Address > 0AM
RAM_8M:
                xchg        eax, ebx
                call        RamBus_Write
                xchg        eax, ebx
                add         bx, INC_8M          ; Next Address
                inc         dh                  ; Next 8M
                dec         cx                  ; More
                jnz         s RAM_8M
```

```
SKIP_8M:
                MEMRD16 ax, RIF_CTL                 ; Fix Up RIF
                and         ax, NOT (RIF_SOUT) AND NOT
                (RIF_STOP_REFRESH) AND NOT (RIF_BANK);

                ;
                ; Compute Size of Memory
                ;
                shr         bx, 2                   ; Position
                dec         bx                      ; encoding is -1
                or          al, bl                  ; Get Number of Bank
                                                    ; Bits

                MEMWR16     RIF_CTL, ax             ; Final Value
;               DELAY
;               MEMWR16 RAC_CTL, 009fh              ; Enable Auto-Calibra
                                                    ; tion
;               DELAY

                mov         al, bl                  ; return it
                ;
                ; Set External Overlay Flag to indicate 9Bit Seen
                ;
                shr         edx, 9h                 ; Position Flag
                mov         bh, dl
                mov         dx, SEQIDX
                mov         al, SP_OPT2
                call        getreg
                or          ah, bh                  ; Set 9th Bit Flag
                setreg

                mov         al, SP_OPT3
                call        getreg
                or          ah, bl                  ; Set Memory Size
                setreg

                call        Init_Chips              ; Find CC Value for
                                                    ; Each Chip

                pop         es
                pop         ds
                ret
```



```
RamBus_Init ENDP

;-----------------------------------------------------------------
;
;             RamBus_Dlay()
;
;             Function:
;                Called from RamBus_Init to slow things down
;
;             Input:
;                None
;             Returns:
;                None
;             Destroys
;                None
;-----------------------------------------------------------------
RamBus_Dlay PROC
              push        ax
              push        cx
              mov         ax, 01h
L1:
              xor         cx, cx
L2:
              out         80h, al
              loop        short L2
              dec         ax
              jnz         short L1
              pop         cx
              pop         ax
              ret
RamBus_Dlay ENDP

;-----------------------------------------------------------------
;
;             RamBus_Ack()
;
;             Function:
;                Wait for a Rambus Ack or Ack3 or Nonexistent
;
;      Register Usage:
;
;             Input:
```

```
;             Rif with Nack bit set and Stop Refresh Set <Very Important>
;            Returns:
;               None
;            Destroys
;               None
;-----------------------------------------------------------------
RamBus_Ack PROC
             push        ax                                  ; Save Some
Registers
             push        cx

             xor         cx, cx                              ; Exit Some Time
RLoop:
             MEMRD16     ax, RIF_CTL                         ; Get it
             and         ax, RIF_ACK_MASK                    ; Isolate Ack
Bits
             cmp         ax, RIF_ACK_MASK                    ; Ack3 ?
             je          s   ACK                             ; We can leave
             cmp         ax, RIF_ACK                         ; Ack ?
             jbe         s   ACK                             ; Below or =
ok...
             loop        RLoop                               ; continue
looking
ACK:
             pop         cx                                  ; Restore for
registers
             pop         ax
             ret
RamBus_Ack ENDP

;-----------------------------------------------------------------
;
;            RamBus_Write()
;
;            Function:
;             Write a Value to the Rambus Data and Transaction Registers
;
;      Register Usage:
;
;            Input:
;               eax = Value for Rambus Data
;               dx = Rambus Transaction
;            Returns:
```



```
;               None
;            Destroys
;               None
;-----------------------------------------------------------------
RamBus_Write PROC NEAR
             MEMWR16     RAMBUS_TRANS, dx
             RNACK
             MEMWR32     RAMBUS_DATA, eax
             RACK
             ret
RamBus_Write ENDP

;-----------------------------------------------------------------
;
;            RamBus_Read()
;
;            Function:
;               Read a Value to the Rambus Data and Transaction Registers
;
;      Register Usage:
;
;            Input:
;               dx = Rambus Transaction
;            Returns:
;               eax = Value form Rambus Data
;            Destroys
;               None
;-----------------------------------------------------------------
RamBus_Read PROC NEAR
             MEMWR16     RAMBUS_TRANS, dx
             RNACK
             MEMRD32     eax, RAMBUS_DATA
             RACK
             ret
RamBus_Read ENDP
;-----------------------------------------------------------------
;
;            Init_Chips()
;
;            Function:
;               Called from RamBus_Init to calibrate Current Control for
;            each Chip
```

```
;
;      Register Usage:
;
;             Input:
;                ax - Device Trans for Chip we are working on
;             Returns:
;                eax - 32 bit value with CC encoded
;             Destroys
;                None
;
;-----------------------------------------------------------------
              PUBLIC Init_Chips
Init_Chips PROC NEAR
              call        Test_Mode
              call        Init_CC                 ; Do zero
              xor         dx, dx                  ; Device Address and Register
                                                  ; Address
IC:
              mov         dl, REG_DEVTYPE         ;
              MEMOR16     RIF_CTL,
RIF_STOP_REFRESH OR RIF_NACK        ; Stop Refresh and Nack
              call        RamBus_Read             ; Read Transaction Register
              test        ax, SIZE_16M            ; 16MB Part
              jz          s P8M                   ;
              inc         dh
P8M:
              inc         dh
              mov         dl, REG_MODE            ; Device Id
              call        RamBus_Read             ; Is it enabled
              and         al, MODE_DE             ; Device Enabled
              jz          s DO_RET                ; Return we are done
              mov         dl, REG_DEVID           ; Device Address
              call        RamBus_Read             ; Read Device Address
              push        eax                     ; Save it
              mov         bx, dx                  ; Save RAMBUS_TRANS
              xor         dh, dh                  ; Address Zero
              xor         eax, eax                ; eax goes to zero
              mov         al, 38h                 ; Set to 14MB
              Call        RamBus_Write            ; Do it
              mov         dx, bx                  ; Set Current to Zero
              xor         eax, eax                ; Zero Address
              Call        RamBus_Write            ; Doit
```



```
                Call        Init_CC                 ; Find "CC"
                mov         dx, REG_DEVID           ; Restore
                pop         eax                     ; to original
                MEMOR16     RIF_CTL,
RIF_STOP_REFRESH OR RIF_NACK                        ; Stop Refresh and Nack
                Call        RamBus_Write            ; position
                mov         dh, 0Eh                 ; 14 MB
                xor         eax, eax                ; Put zero's chip
                Call        RamBus_Write            ; Back
                mov         dx, bx                  ; Restore
                jmp         s IC                    ; Continue

DO_RET:
                MEMAND16 RIF_CTL,
                NOT(RIF_STOP_REFRESH)               ; Enable Refresh
                ret
Init_Chips ENDP

;-----------------------------------------------------------------
;
;               Init_CC()
;
;               Function:
;                  Called from Init_Chips to calibrate Current Control
;
;       Register Usage:
;
;       ax - Temporary
;       bx - Accumulator
;       cx - Number of Passes
;       dx - Mode Register
;
;               Input:
;
;               Returns:
;
;               Destroys
;
;
;-----------------------------------------------------------------
                PUBLIC Init_CC
Init_CC PROC NEAR
```

```
                push        bx
                xor         bx, bx                  ; Bx goes to zero
                mov         cx, PASSES              ; Number of Passes
                xor         dh, dh                  ; Device Address = 0
                mov         dl, REG_MODE            ; Mode Register
FCC:
                call        Find_CC                 ; Find Right CC Value
                add         bx, ax                  ; Accumulate in bx
                loop        s FCC                   ; PASSES times?
                shr         bx, 2                   ; Divide by Passes
                mov         ax, bx                  ;
                call        Encode                  ; Encode it
                or          al, MODE_DE OR
                MODE_X2 OR MODE_AS OR MODE_CE


                MEMOR16     RIF_CTL,
                RIF_STOP_REFRESH OR RIF_NACK        ; Stop Refresh and Nack
                call        RamBus_Write                        ; Write the Data
                MEMAND16 RIF_CTL,
                NOT(RIF_STOP_REFRESH)               ; Enable Refresh
                pop         bx
                ret
Init_CC ENDP
;---------------------------------------------------------------
;
;               Find_CC()
;
;               Function:
;                  Called from Init_CC to calibrate Current Control
;
;       Register Usage:
;
;                  al - Current Value
;                  bx - PrevPass
;        cx - Intermediate Value
;        dx - Return value from Test_CC
;        si - Accumulator
;
;                  Input:
;                     None
;                  Returns:
;                     al - With value to use in Auto Calibration
;
```



```
;             Destroys
;                si
;
;-----------------------------------------------------------------
              PUBLIC Find_CC
Find_CC PROC NEAR
              push        bx                    ; Save some registers
              push        cx
              push        dx

              xor         al, al                ; Set Current Value to Zero
              xor         bx, bx                ; PrevPass goes to zero
              xor         si, si                ; Accumulator goes to zero
F1:
              cmp         al, 63                ; If more then max value
              jbe         s F2                  ; No
              xor         ax, ax                ; return 0
              jmp         s F_Exit

F2:
              call        Test_CC               ; Test Current Value
              cmp         dl, 0                 ; if pass > 0
              jle         s F3                  ; None Passed

              xor         dh, dh                ; dh is not needed
              mov         cx, dx                ; Current Value
              xor         ah, ah                ; Nor is ah needed
              sub         cx, bx                ; Current Pass - Previous
Pass
              xchg        cx, ax                ; Swap for Mult
              imul        ax, cx                ; Index * (CP - PP)
              xchg        cx, ax                ; Swap again
              add         si, cx                ; Add to Accumulator
              mov         bx, dx                ; Update Previous Pass

              cmp         dl, 80                ; 10 passes at 8 bits per
pass?
              je          s F4                  ; then exit
F3:
              inc         al                    ; Increment current value
              jmp         s F1                  ; Back to top
F4:
```

```
                sub         si, 40                  ; i_sumv - 40
                mov         ax, si                  ;
                imul        ax, MULT                ; i_ccvalue = MULT * i_sumv -
40
                call        CMtoAuto                ; Convert to Auto

F_Exit:
                pop         dx                      ; Restore some registers
                pop         cx
                pop         bx
                ret
Find_CC ENDP

;---------------------------------------------------------------------
;
;               Test_CC()
;
;               Function:
;                  Called from RamBus_Init to calibrate Current Control
;
;       Register Usage:
;                eax - Temporary
;                bx - Temporary
;         cx - Loop Counter
;         dl - Accumulator
;         es:di - BF00 Frame Buffer
;               Input:
;                 al - Value to Calibrate
;               Returns:
;                 dl - Number of bits in ten passes that were set
;               Destroys
;                 di
;                 es
;
;---------------------------------------------------------------------
                PUBLIC Test_CC
Test_CC PROC NEAR
                push        ax                      ; Save some registers
                push        bx                      ;
                push        cx
                push        es
```



```
                mov         bx, 0BF00h              ; Set es to BF00
                mov         es, bx

                ;
                ; Manual Calibration
                ;

                call        Encode                  ; Encode Value
                mov         dx, REG_MODE            ; Set dx to right registers
                or          al, MODE_DE
OR MODE_X2 OR MODE_AS                               ; Set Up
                MEMOR16     RIF_CTL,
RIF_STOP_REFRESH OR RIF_NACK                        ; Stop Refresh and Nack
                call        RamBus_Write
                MEMAND16 RIF_CTL,
NOT(RIF_STOP_REFRESH)                               ; Enable Refresh

                mov         cx, READ_TSIZE          ; Number of Times to Try
                xor         dl, dl                  ; Setup Counter
                xor         eax, eax                ; eax = 0
                not         eax                     ; eax = -1

W1:
                xor         di, di                  ; Set di to zero
                stosd                               ; Save -1
                stosd                               ; Save -1
                dec         di                      ; Pointer to byte 6
                push        eax
                mov         al, es:[di]             ; Correct ?
                call        Bit_Count               ; Find Number of bits set
                add         dl, al                  ; Number Found
                pop         eax
                loop        W1

                pop         es                      ; Restore registers
                pop         cx
                pop         bx
                pop         ax
                ret
Test_CC ENDP

;-----------------------------------------------------------------
;
```

```
;             CMtoAuto()
;
;             Function:
;                Called from Find_CC to calibrate Auto Current Control
;
;       Register Usage:
;        ax - temporary
;        bx - i_ccvalue
;        cl - i_accval
;        si - i_delccval
;        di - i_minccval
;        dx - temporary
;
;             Input:
;             ax - value to use
;
;             Returns:
;
;             Destroys
;
;
;---------------------------------------------------------------
              PUBLIC CMtoAuto
CMtoAuto PROC NEAR
              xor         ebx, ebx                ; Set Top half of ebx to zero
              mov         bx, ax                  ; Save Current Value

              xor         esi, esi
              mov         si, 51200               ; Initialize Some Variables
              mov         cx, 0FFFFH              ; cx=-1
              xor         di, di
C1:
              inc         cx
              cmp         cl, 63                  ; All done??
              jbe         s CONT
              xor         ax, ax                  ; return 0
              ret
CONT:
              mov         dx, REG_MODE            ; Set up for Rambus
Reads/Writes
              mov         al, cl                  ; Get Next Value
              call        Encode                  ; Encode Value
```



```
                or          al, MODE_CE OR
MODE_DE OR MODE_X2 OR MODE_AS       ; Set Up
                MEMOR16     RIF_CTL,
RIF_STOP_REFRESH OR RIF_NACK        ; Stop Refresh and Nack
                call        RamBus_Write
                call        RamBus_Read
                call        RamBus_Read
                MEMAND16 RIF_CTL,
NOT(RIF_STOP_REFRESH)               ; Enable Refresh
                call        Decode
                xor         ah, ah                  ; Only want lower byte
                imul        eax, 800                ;
                mov         edx, eax
                sub         eax, ebx
                jnc         s NO_CARRY
                neg         eax

NO_CARRY:
                cmp         eax, esi                ; i_dccv < i_delccval
                jge         s C2
                mov         esi, eax                ; Update i_delccval = i_dccv
                mov         di, cx                  ; i_minccval = i_accval

C2:
                cmp         edx, ebx                ; i_readccval >= i_ccvalue
                jl          s C1

                mov         al, cl
                cbw
                add         ax, di
                shr         ax, 1
                ret
CMtoAuto ENDP

;-----------------------------------------------------------------
;
;               Encode()
;
;               Function:
;                  Called from RamBus_CC to Encode current Calibration
;
;       Register Usage:
;               ah          -  Number of Times to Loop
```

```
;             al          -  Value to encode
;             ebx         -  Accumulated Value
;             ch          -  And Mask
;             cl          -  Shift Mask
;             edx         -  Single bit Value
;             ds          -  Pointer to code segment
;             si          -  Offset to Encode Table
;
;             Input:
;                al - has 6 bit value to encode
;             Returns:
;                eax - 32 bit value with CC encoded
;             Destroys
;                None
;-----------------------------------------------------------------
              ASSUME DS:cgroup
              PUBLIC Encode
Encode PROC NEAR
              push        ebx                   ; Save a few registers
              push        cx                    ;
              push        edx                   ;
              push        si                    ;
              push        ds                    ;

              mov         ah, ENCODE_CNT        ; Number of Words to use
              mov         si, OFFSET ENCODE_TABLE
              mov         bx, cs                ; Code Segment
              mov         ds, bx                ; Code Segment
              xor         ebx, ebx              ; Clear Accumulator
              xor         al, CMASK             ; Data Mask
E1:
              xor         edx, edx              ; Clear for Single Bit
              xchg        ax, cx                ; Save ax
              lodsw                             ; Load new mask and shift
              xchg        cx, ax                ; Restore ax
              mov         dl, al                ; Get cc
              and         dl, ch                ; And Mask
              shl         edx, cl               ; Shift to Position
              or          ebx, edx              ; Accumulate
              dec         ah                    ; Any more
              jnz         s E1                  ; ???
              mov         eax, ebx              ; Return Value
```



```
                pop         ds                      ; Restore some registers
                pop         si
                pop         edx
                pop         cx
                pop         ebx
                ret
Encode ENDP

;-----------------------------------------------------------------
;
;               Decode()
;
;               Function:
;                  Called from RamBus_CC to Encode current Calibration
;
;       Register Usage:
;               ebx         -  Value to  decode
;               ah          -  Number of Times to Loop
;               al          -  Temporary
;               ebx         -  CC Value
;               ch          -  Accumulated Value
;               cl          -  Shift Mask
;               edx         -  Single bit Value
;               ds          -  Pointer to code segment
;               si          -  Offset to Decode Table
;
;               Input:
;               eax         -  Value to  decode
;               Returns:
;               al -  Value decode
;               Destroys
;                  None
;-----------------------------------------------------------------
                ASSUME DS:cgroup
                PUBLIC Decode
Decode PROC NEAR
                push        ebx                     ; Save a few registers
                push        cx                      ;
                push        edx                     ;
                push        edi
                push        si                      ;
```

```
                push        ds                      ;

                mov         si, OFFSET DECODE_TABLE
                mov         bx, cs                  ; Code Segment
                mov         ds, bx                  ; Code Segment
                mov         ebx, eax                ; Value to Decode
                mov         dh, DECODE_CNT          ; Number of Words to use
                xor         ch, ch                  ; Clear Accumulator
                xor         dl, dl                  ; Clear Accumulator
D1:
                mov         edi, ebx                ; Get CC
                lodsd                               ; Load new mask
                and         edi, eax                ; Mask it
                lodsb                               ; Shift Value
                mov         cl, al                  ; Shift
                shr         edi, cl                 ; Shift it
                or          dx, di                  ; Get New Value
                or          ch, dl                  ; Accumulate
                dec         dh                      ; More ??
                jnz         s D1
                mov         al, ch                  ; New Value

                pop         ds                      ; Restore some registers
                pop         si
                pop         edi
                pop         edx
                pop         cx
                pop         ebx
                ret
Decode ENDP

;-----------------------------------------------------------------
;
;               Bit_Count()
;
;               Function:
;                  Count number of bits set in a byte
;
;       Register Usage:
;               ah          -  Bit Position
;               al          -  Byte to Count
;               ch          -  Accumulator
```



```
;
;             Input:
;             al -  Byte to Count
;             Returns:
;             al -  Bit Count
;             Destroys
;                None
;-----------------------------------------------------------------
              PUBLIC Bit_Count
Bit_Count PROC Near
              push        bx
              push        cx                    ; Save cx
              mov         bx, 7h                ; Test all 7 bits
              xor         cl, cl                ; Accumulator
              or          al, al                ; Any bits at end
              jz          s EO                  ; Not just leave
K1:
              bt          ax, bx                ; Bit Set
              adc         cl, 0                 ; If so add carry in
              dec         bx                    ; Decrement ah
              jnz         s  K1                 ; ah = 0?
              bt          ax, bx                ; do zero bit
              adc         cl, 0                 ; add carry
EO:
              mov         al, cl                ; Get Result
              pop         cx                    ; restore cx
              pop         bx
              ret
Bit_Count ENDP

SCR_TBL:
              dw   0101h  ; SR01<-01
              dw   0F02h  ; SR02<-0F
              dw   0003h  ; SR03<-00
              dw   0E04h  ; SR04<-0E
              dw   0107h  ; SR07<-01
S_COUNT = ($ - SCR_TBL) SHR 1

GFX_TBL:
              dw   0000h  ; GR00<-00
              dw   0001h  ; GR01<-00
              dw   0002h  ; GR02<-00
```

```
                dw   0003h  ; GR03<-00
                dw   0004h  ; GR04<-00
                dw   4005h  ; GR05<-40
                dw   0D06h  ; GR06<-0D
                dw   0F07h  ; GR07<-0F
                dw   0FF08h ; GR08<-FF
G_COUNT = ($ - GFX_TBL) SHR 1


;-----------------------------------------------------------------
;
;               Test_Mode()
;
;               Function:
;                  Used to put the VGA into a 8 bit packed pixel mode
;               for RAMBUS
;             Test.  This allows a 1-1 mapping between the linear
;               frame buffer and the VGA aperature
;
;       Register Usage:
;
;               Input:
;               None
;               Returns:
;               None
;               Destroys
;               ax
;               cx
;               dx
;               si
;-----------------------------------------------------------------
                PUBLIC Test_Mode
Test_Mode PROC NEAR
                push        ds
                mov         cx, cs
                mov         ds, cx

                mov         dx, SCRIDX
                mov         cx, S_COUNT
                mov         si, OFFSET SCR_TBL
S1:
                lodsw
```



```
                setreg
                loop        s S1

                mov         dx, GFXIDX
                mov         cx, G_COUNT
                mov         si, OFFSET GFX_TBL
G1:
                lodsw
                setreg
                loop        s G1
                mov         al, 02fh
                mov         dx, 03c2h
                out         dx, al
                pop         ds
                ret
Test_Mode ENDP


VGA_Segment   ENDS
                END
```

5.1.2 Frame Buffer

The CL-GD546X has a frame buffer with both VGA and linear apertures. The linear aperture is always visible. The programmer can write to and read the physical address of the linear frame buffer aperture by the BASE_ADDRESS_1 register, in either memory-mapped I/O space at 314h or PCI space at offset 14h. In a PCI system, this task is typically allocated to the PCI system BIOS.

The EXT_DISP bit in the VS_CONTROL register tells the display pipeline whether to use the VGA registers or the pixel format register to determine the memory layout. The EXT_DISP bit is used in conjunction with bit SR07[0]. These two bits are programmed as follows:

```
IF BPP < 8 then
            VS_CONTROL.EXT_DSP = 0
            SR07[0] = 0
            Use VGA Controller to Access Frame Buffer
else if BPP > 8 then
            VS_CONTROL.EXT_DSP = 1
            SR07[0] = 0
            Set Pixel Format to set desired depth
            Use Linear Frame Buffer
else if Amount of Display Memory for Mode <= 1MB and SVGA
            Compatibly is Desired then
            VS_CONTROL.EXT_DSP = 0
            SR07[0] = 1
            Use VGA Controller to Access Frame Buffer
else
            VS_CONTROL.EXT_DSP = 1
            SR07[0] = 0
            Set Pixel Format to set desired depth
            Use Linear Frame Buffer
endif
```

5.1.3 Other Registers of Interest

The 2D engine and video pipeline are programmed in certain register fields. Table 5-1 lists the register fields and their locations.

Table 5-1. 2D Engine Programmable Register Fields

Fields	Registers
Bits-per-pixel	Format, Control
Interleave	Display Threshold and Tiling, TileCtrl
Pixel Format	Format, LnCtrl
Tile versus Linear Memory Addressing	Display Threshold and Tiling, Control
Tile Size	Display Threshold and Tiling, Control

Table 5-1. 2D Engine Programmable Register Fields *(cont.)*

Fields	Registers
Tiles-per-line	Display Threshold and Tiling, TileCtrl
Threshold	Display Threshold and Tiling

To program the registers in Table 5-1, follow the procedure listed below. If the BIOS is used to set a mode, this is procedure done automatically.

1) Choose the bits-per-pixel (8, 16, 24, or 32). Program into the CONTROL.BPP and FORMAT.Pixel_Depth register fields. Also, choose a pixel format (typical values are 8 bpp for palletized, and 16, 24, and 32 bpp for RGB). These values are programmed into the FORMAT.Pixel_Format and LNCNTL.Pixel_Format register fields. Note that the encoded values are not the same for these fields. For 16 bpp, the LNCNTL.Pixel_Format is 5:6:5 or 2.

NOTE: SR07[0] must be zero to change FORMAT register.

2) Choose a set of screen extents (dX and dY).

3) Multiply the dX, X extent by the number of bytes-per-pixel to get the number of bytes-per-scanline.

4) Choose a 'Pitch in Bytes', from the Table 5-2, that is greater than or equal to the number of bytes-per-line calculated. Optimal pitches are given in Table 5-2. If a chosen value is equal, all of the off-screen memory is in one contiguous rectangle at the bottom of the screen. This simplifies off-screen memory management.

Table 5-2. CL-GD546X Display Resolutions

Screen Size, bpp and Pitch					Memory Size and Pitch				Concurrent Video	
dX	dY	bpp	Minimum Available Pitch (in bytes)	Optimum Tile Size	1 Mbyte	2 Mbytes	3 Mbytes	4 Mbytes	YUV	RGB
640	480	8	640	128	640				AccuPak 4:2:2	8, 555, 565
640	480	16	1280	128	1280				4:2:2	555, 565
640	480	24	2048	256		2048			4:4:4	888
640	480	32	2560	256		2560			4:4:4	888x
800	600	8	1024	128	1024				AccuPak 4:2:2	8, 555, 565
800	600	16	1664	128	1664				4:2:2	555, 565
800	600	24	2560	256		2560			4:4:4	888
800	600	32	3328	256		3328			4:4:4	888x

Table 5-2. CL-GD546X Display Resolutions *(cont.)*

Screen Size, bpp and Pitch					Memory Size and Pitch				Concurrent Video	
dX	dY	bpp	Minimum Available Pitch (in bytes)	Optimum Tile Size	1 Mbyte	2 Mbytes	3 Mbytes	4 Mbytes	YUV	RGB
1024	768	8	1024	128	1024				AccuPak 4:2:2	8, 555, 565
1024	768	16	2048	128		2048			4:2:2	555, 565
1024	768	24	3072	256			3072		4:4:4	888
1024	768	32	4096	256			4096		4:4:4	888x
1280	1024	8	1024	128		1280			AccuPak 4:2:2	8, 555, 565
1280	1024	16	2048	128			2560		4:2:2	555, 565
1600	1200	8	1664	128		1664			4:2:2	8, 555, 565

5) Program the CONTROL.TILE_SIZE, DISPLAY_THRESHOLD_AND_TILING.TILING and DISPLAY_THRESHOLD_AND_TILING.WIDE_TILE register fields with the tile size in Table 2-2.

6) Write the corresponding value in the TILE_CTRL column to register fields TILE_CTRL.TILES_PER_LINE and DISPLAY_THRESHOLD_AND_TILING.TILES_PER_LINE. The DISPLAY_THRESHOLD_AND_TILING.FIFO depends on the PCLK, MCLK, and the bits-per-pixel. The minimum and maximum values are calculated as described below, along with the average of the two values used. Program the FIFO threshold value midway between the minimum and maximum value.

Minimum FIFO threshold is chosen to prevent FIFO underrun, as shown in Equation 5-1.

$$Minimum = \frac{17}{64} \bullet PCLK \bullet \frac{bpp}{64} + 1$$

Equation 5-1

where

17 is the worst-case Rambus latency (in MCLKs) from when the FIFO makes the request, until the first word of data is in the FIFO.

CLK is the programmed MCLK frequency.

PCLK is the programmed PCLK frequency.

bpp is the programmed number of bits-per-pixel. *64* is the number of bits in a FIFO word.

1 is one extra FIFO word since the same word cannot be simultaneously read or written.

The result is rounded up to the nearest integer, due to the fact that the current word may already have some pixels shifted out.

Maximum FIFO threshold is chosen to prevent FIFO overrun, as shown in Equation 5-2.

$$Maximum = \left[48 - (16,32) + \frac{\{9 + (16,32)\}}{CLK} \bullet PCLK \bullet \frac{bpp}{64}\right]$$

Equation 5-2

where
48 is the number of words in the FIFO.
{16|32} is the number of FIFO words read (either 16 or 32, as programmed in the Tiling field).
9 is the best-case Rambus latency (in MCLKs) from when the FIFO makes the request until the first word of data is in the FIFO.
The next *{16|32}* is the number of MCLKs to read 16 or 32 FIFO words.
CLK is the programmed MCLK frequency.
PCLK is the programmed PCLK frequency.
bpp is the programmed number of bits-per-pixel and 64 is the number of bits in a FIFO word.

For example, to calculate for 1024 × 768 × 8 bpp, 75 Hz (75-MHz PCLK), 62.5-MHz MCLK, tiling = 00 (128 byte packets — 16 FIFO words) use Equation 5-3 and Equation 5-4.

$$Minimum = \left[\frac{17}{62.5e6} \bullet 75e6 \bullet \frac{8}{64}\right] + 1 = 4$$

Equation 5-3

$$Maximum = \left[48 - \left(16 + \frac{(9 + 16)}{62.5e6}\right) \bullet 75e6 \bullet \frac{8}{64}\right] = 28$$

Equation 5-4

5.1.4 Memory Tile Interleave Setup

For optimal system performance, the programmer should set up Memory tile interleaving. The goal is to reduce the number of page breaks or tile crossings in a given graphics operation. Since each of the four tiles are mapped into a different memory bank, four-way interleaving means that crossing tile boundaries at a four-tile intersection performs in the optimal manner. When using two-way interleave, an operation that crosses two tiles should not incur a memory penalty after the first two accesses in the respective tiles. As specified in Table 5-3, it is best to choose the highest interleave value possible for a given memory configuration. Program the values into the TILECTL.BANK_INTERLEAVE and DISPLAY_THRESHOLD_AND_TILING.INTERLEAVE registers.

Table 5-3. Interleave Values for Memory Configurations

Mbytes of Frame Buffer Memory	Interleave Choices Preferred	Interleave Choices Allowable	TILE_CTRL.BANK_INTERLEAVE
1	1 way	1 way	0
2	2 way	1 and 2 way	1
3	1 way[a]	1 way[a]	0
4	4 way	1, 2, and 4 way	2
5	1 way[a]	1 way[a]	0
6	1 way	1 and 2 way	0
7	1 way[a]	1 way[a]	0
8	4 way	1, 2, and 4 way	2

[a] Depending on resolution, how much off-screen memory is required, and what performance improvement is obtained by going to larger interleave, the programmer need to use higher interleaves and 'waste' memory. For example, with 5 Mbytes at 640 × 480 × 8, the programmer may use four-way interleave for maximum performance and 'waste' the fifth Mbyte. Optionally, this might be used for 3 Mbytes at two-way interleave for low resolutions and color depths, and one-way interleave for 1024× 768 × 24 mode.

The CONTROL, TIMEOUT, and TILE_CTRL register setup is summarized in Table 5-4.

Table 5-4. CONTROL and TIMEOUT Register Setup

Register	Initial Value	Notes
CONTROL.BPP	0,1,2,3	Select 8, 16, 24, 32 bpp.
CONTROL.CNTL_TAG	NA	Read only
CONTROL.FLUSH_PIPE	0	
CONTROL.FLUSH_RDFIFO	0	
CONTROL.FLUSH_WRFIFO	0	
CONTROL.HALT_FIFO	0	
CONTROL.HALT_PIPE	0	
CONTROL.SWIZ	0	Do not swizzle
CONTROL.TILE_SIZE	0,1,3	128, 256, 2048 tile size. Select in concert with pitch selection in TILE_CTRL.
TILE_CTRL.INTERLEAVE	0,1,2	Select 1 way, 2 way, or 4 way (seeTable 5-3).
TILE_CTRL.WIDPIT	[a]	
TIMEOUT	0Fh	Disable

[a] Value from CL-GD546X display resolutions in Table 5-2.

5.1.5 Differences from Standard VGA

The CL-GD546X has more registers than standard VGA. The registers are divided into two parts: Indexed I/O registers and Memory-Mapped I/O registers. Table 5-5 describes the additional indexed I/O registers.

Table 5-5. CL-GD546X VGA Indexed I/O Registers

Sequencer Registers	
SR07[0]	Allows true 8-bit Packed Pixel mode. Setting this bit allows each pixel to be packed instead of planar.
SR09, SR0A, SR14, SR15	Scratch Pad registers. These four registers are reserved for use by the BIOS. These registers store the BIOS variables that cannot reside in conventional memory.
SRB-SRE/SR1B–SR1E	Programs the different frequencies for video clocks.
SR18, SR19, SR1A	Tests registers for device debug and testing.
CRT Controller Registers	
CR19	Centers odd scanlines in interlace modes.
CR1A	Miscellaneous Control register to extend Vertical Blank End and Horizontal Blank End, DPMS support, enable double buffered display start address and interlace.
CR1B	Controls the Blank Extended bits, Enable/Disable Borders, Extend Offset, Display Start, and Memory Wrap.
CR1D	Extends Screen Start A address and offset.
CR1E	Extends Horizontal Total, Horizontal Display End, Horizontal Blank Start, Horizontal Sync Start, Vertical Total, Vertical Display End, Vertical Blank Start, and Vertical Sync Start.
CR22	Reads the four graphics controller data latches.
CR24	Reads the attribute controller toggle.
CR26	Reads the attribute controller index and Video Enable bit.
Graphics Controller Registers	
GR9	Provides access to 1 Mbyte of memory with 4-Kbyte granularity. This provides address bits A[19:12] and allows the programmer to address beyond the 256 Kbytes allowed by VGA.
GRA	Provides access to 1 Mbyte of memory with 4-Kbyte granularity. This provides address bits A[19:12] and allows the programmer to address beyond the 256 Kbytes allowed by VGA.
GRB[0]	Chooses between GR9 and GRA to provide the offset. This allows the programmer two windows for copy data, where one or both of the windows is beyond 256 Kbytes. For GR9 offset use GRB[0] = 0; for GRA offset use GRB[0] = 1 and SA15 = 1.

The important registers and bits in the CL-GD546X to set up to ensure standard VGA compatibly mode are:

- Display Threshold and Tiling
- Sequence – SR07[0]
- VS_Control.EXT_DSP
- the Pixel Format Gamma Correction bit, which allows the DAC to be 6 or 8 bits in SVGA modes

5.2 CRTC Programming

The following section discusses the programming of registers in the CRTC (CRT Controller). These registers control the video timing.

5.2.1 VESA® Timing Specifications

The video timing for most modes is defined in the VESA Monitor Timing specifications, available through VESA. Wherever possible, Cirrus Logic uses VESA timing. Table 5-6 indicates the resolutions and refresh rates defined by VESA as of May 1995.

Table 5-6. VESA® Monitor Timing Specifications

Refresh/ Resolution	Interlaced	56	60	65	70	72	75	80	85
640 × 350			I[a]		I				VP[b]
640 × 400			I		I				VP
640 × 480			I			VO[c]	VS[d]		VP
800 × 600		VS	VS			VO	VS		VP
1024 × 768	I		VS		VS		VS	VP	VP
1152 × 864			VS		VP		VP		VP
1280 × 960	VP		VP				VP		VP
1280 × 1024	VP		VP				VS		VP
1600 × 1200	VP		VP	VP	VP		VP	VP	VP

[a] 'I' indicates industry standard.

[b] 'VP' indicates VESA proposal.

[c] 'VO' indicates obsolete VESA standard.

[d] 'VS' indicates VESA standard.

NOTE: The appearance of any notation in the above table does not mean that the Cirrus Logic BIOS supports the corresponding specification. This table is provided as a reference only.

5.2.2 CRTC Timing

Figure 5-1 illustrates how CRTC timing is generated. There are two sets of timing: horizontal and vertical. When these sets are combined, an orthogonal raster is produced. These timings are similarly generated.

Horizontal timing is generated by counting *character clocks* (this term is derived from character graphics: the pixel clock divided by eight). The vertical timing is generated by counting *scanlines*.

Each counter counts from zero to the respective maximum value, then resets to zero to start again. The zero value of the horizontal counter corresponds to the first – left-most – pixel of active video; the zero value of the vertical counter corresponds to the top scanline on the screen.

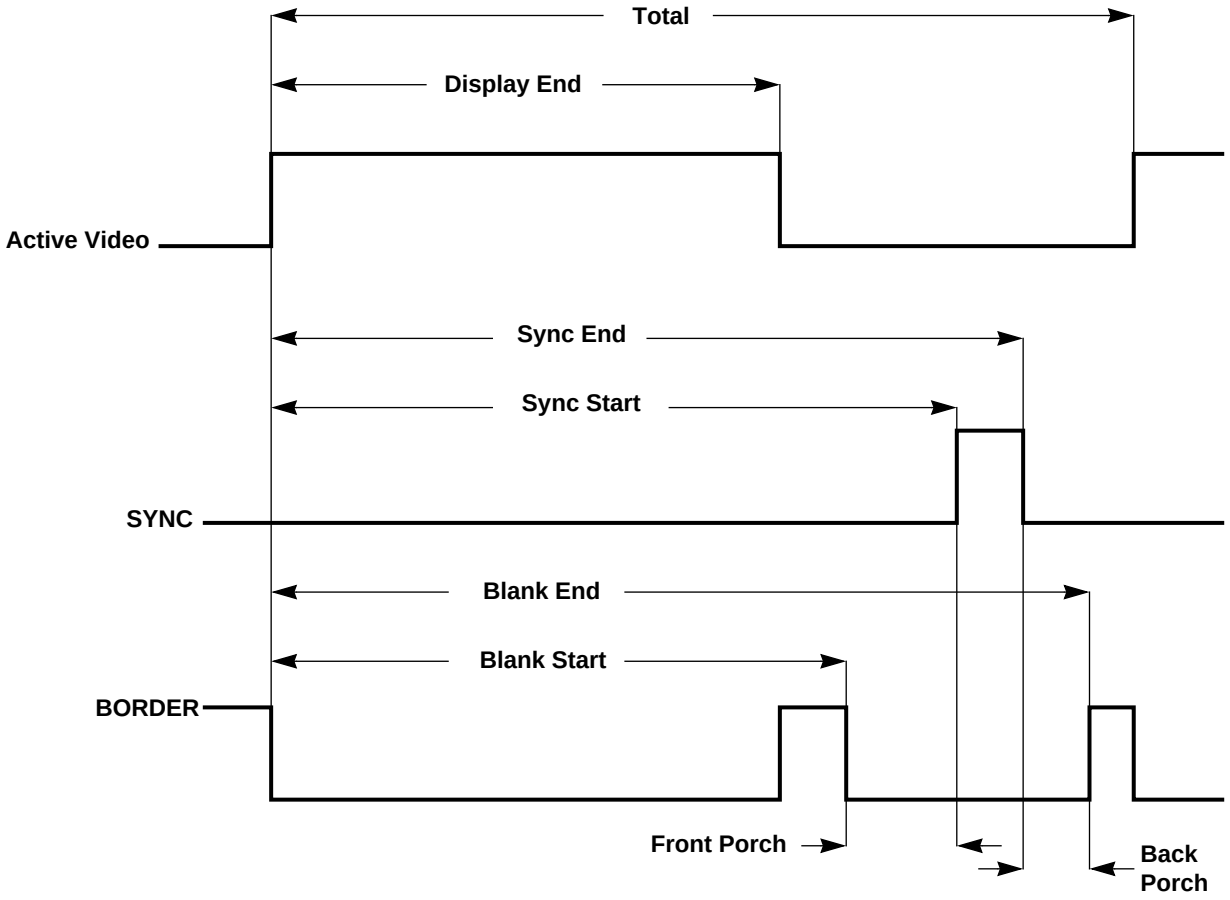

Figure 5-1. CRTC Timing

As either counter increases from zero – counting either pixels or scanlines – it arrives at the values programmed for the various events in the raster, as shown in Table 5-7. Many of the values do not fit into a single 8-bit register. The extensions, assigned either as part of the VGA specification or by Cirrus Logic, are presented in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*. For Sync End and Blank End for both horizontal and vertical, fewer than the total number of bits in the respective counter are compared. This results in the width limitations on sync pulses and blanking periods.

Table 5-7. CRTC Registers

Event	Horizontal	Vertical	Next Period
Display End	CR1	CR12	Border or front porch
Blank Start	CR2	CR15	Front porch
Sync Start	CR4	CR10	Sync
Sync End	CR5	CR11	Back porch
Blank End	CR3	CR16	Border or active video
Total	CR0	CR6	Active video (next scanline or frame)

5.2.3 Programming VCLK

The fundamental timing source for the CRTC timing is the pixel clock, which comes from the VCLK synthesizer or from the DCLK pin. When the VCLK synthesizer is used, the frequency is determined by the programming of one of four register pairs. Use Equation 5-5 when the post-scaler is '0'.

$$VCLK = RCLK \bullet \frac{Numerator}{Denominator}$$

Equation 5-5

Use Equation 5-6 when the post-scaler is '1'.

$$VCLK = RCLK \bullet \frac{Numerator}{\langle Denominator \bullet 2 \rangle}$$

Equation 5-6

where,
RCLK is the reference frequency, normally 14.31818 MHz.
Numerator is a 7-bit value, programmed into SR1B–SR1E[6:0].
Denominator is a 7-bit value, programmed into SRB–SRE[7:1].

The post-scalar value is a single bit to indicate if the synthesizer output is divided by two. This bit is SRB–SRE[0]. Refer to the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)* for register information. The register pair is selected in MISC[3:2].

The programmed VCLK is multiplied by two at the RAMDAC to support extended modes with a pixel rate of greater than 85 MHz. Select this mode by programming Graphics/Video Format[14] to '1'. The horizontal timing is 8 pixels.

5.2.4 Programming BCLK

BCLK is the Rambus clock and is directly programmed by specifying an appropriate value into the BCLK Multiplier register (MMI/O offset 8Ch). This is shown in the equation below. The BCLK Multiplier may be any value from 7 to 22 (decimal). The nominal value is 18d, returning a BCLK frequency of 257.7 MHz, with a nominal RCLK input of 14.31818 MHz.

$$BCLK = RCLK \bullet BCLKMultiplier$$

Equation 5-7

5.2.5 Non-Standard Timing Parameters

The VESA standards, referenced in Section 5.2.1, provide precise timing details for standard monitors using standard resolutions. There are two situations in which the standard values cannot be used. It may be necessary to program a non-standard resolution, or program a standard resolution for a non-standard monitor.

The programmer can use the following general approach:

- Decide on the horizontal timing. If the monitor manufacturer cannot provide guidelines for the horizontal total and blank timing, select another monitor.

$$HorizontalActiveTime = HorizontalTotal - HorizontalBlanking$$

Equation 5-8

- The horizontal active time can be calculated by subtracting the horizontal blank from the horizontal total. Dividing horizontal active by the number of pixels-per-scanline yields the pixel period.

$$PixelPeriod = \frac{HorizontalActive}{(PIxels)/(Scanline)}$$

Equation 5-9

- The pixel frequency is the reciprocal of the pixel period.

$$VCLK = \frac{1}{PixelPeriod}$$

Equation 5-10

The VCLK synthesizer is programmed using one of the above equations. For most frequencies, a number of solutions yield the desired frequency. In general, the best results are obtained by choosing a solution with both the numerator and denominator in the middle of their respective ranges. If possible, use the post-scalar especially for frequencies below ~30 MHz. If the pixel clock is above 135 MHz, select clock doubling by programming Graphics/Video Format[14] (MMI/O C0h) to '1' for 8-bpp modes.

Even with these considerations, there are usually a number of sets of values that produce the desired frequency. The choice among these sets of values is generally determined empirically.

The horizontal counter counts with a resolution of 8 pixels (even with pixel doubling enabled). All horizontal timing is in terms of this counter. The Horizontal Display End and Horizontal Total values are determined, respectively, by the pixels-per-scanline and horizontal period.

If a horizontal border is not used, program Horizontal Blank Start to correspond to Horizontal Display End, and program the Horizontal Blank End to correspond to Horizontal Total. If a horizontal border is used, Horizontal Blank Start follows Horizontal Display End, and Horizontal Blank End occurs before Horizontal Total. The differences are the desired border width.

Program the horizontal sync pulse next. The horizontal sync pulse generally begins near the beginning of horizontal blanking (Horizontal Blank Start). The polarity of the sync pulse is programmed in the MISC register. Typically, the sync width is not important; most monitors require only the leading edge. The position of the sync pulse can be adjusted to position active video on the monitor. Forcing the sync pulse to occur earlier, moves the video to the left.

The vertical parameters are selected the same way, except in terms of scanlines.

5.3 Hardware Cursor

This section describes the 64 × 64 hardware cursor (mouse pointer) that the CL-GD546X can support in all color depth packed-pixel graphics modes. The hardware cursor replaces the software mouse pointer commonly used by GUI applications. The hardware cursor eliminates the need for application software to save and restore the screen data as the mouse position changes.

The application software typically initializes the hardware cursor once. From that time, it only needs to update the cursor (xy) position to move the cursor on the screen. The hardware cursor offers a smooth moving mouse pointer with improved performance.

Multiple hardware cursor patterns are loaded into off-screen display memory, allowing application programs to quickly select between them.

5.3.1 Hardware Cursor Operation

To operate the hardware cursor, the CL-GD546X has a set of Memory-Mapped registers that enable/disable the hardware cursor. These registers specify the location of the hardware cursor pattern in off-screen memory, position the hardware cursor on the screen, and control the offset within the pattern displayed at the current position on the screen. The Memory-Mapped registers related to the operation of the hardware cursor are listed in Table 5-8. I/O ports related to the operation of the hardware cursor are listed in Table 5-8. The I/O ports are applied when programming the hardware cursor palette entries. Otherwise, all hardware cursor control is done by the Memory-Mapped registers. The offset column in Table 5-8 is relative to the start of the linear address where the CL-GD546X Memory-Mapped registers are mapped.

Table 5-8. Hardware Cursor Memory-Mapped Registers

Register Name	Offset	Size	Bit	Descriptions
PALETTE_STATE	0B0h	8 bits	7	DAC power-down
			6	External DAC
			5:4	Reserved
			3	Access cursor color
			2	Read mode
			1:0	Palette state
CURSOR_X	0E0h	16 bits	15:12	Reserved
			11:0	CURSOR_X
CURSOR_Y	0E2h	16 bits	15:12	Reserved
			11:0	CURSOR_Y
CURSOR_PRESET	0E4h	16 bits	15	Reserved
			14:8	X_PRESET
			7	Reserved
			6:0	Y_PRESET
CURSOR_CONTROL	0E6h	16 bits	15:13	Reserved
			12	COMPOSITE_SYNC
			11:9	Reserved
			8	STEREO_ENABLE
			7:1	Reserved
			0	CURSOR_ENABLE
CURSOR_LOCATION	0E8h	16 bits	15	Reserved
			14:2	CURSOR_OFFSET
			1:0	Reserved

NOTE: The I/O ports for palette entries listed in Table 5-9 are also addressable with MMI/O.

Table 5-9. I/O Ports Used for Palette Entries

Reg Name	I/O Port	Offset
PALETTE_READ_ADDR	03C7h	A4
PALETTE_WRITE_ADDR	03C8h	A8
PALETTE_DATA	03C9h	AC

Use of the hardware cursor is enabled by setting CURSOR_CONTROL[0] to '1'.

The hardware cursor is stored in off-screen display memory as two monochrome bit masks. Each mask is 64 bits wide by 64 bits high (512 bytes per mask). The first bit mask is the Plane 0 mask; the second bit mask is the Plane 1 mask. Table 5-10 shows the output displayed by the CL-GD546X hardware cursor. In the Microsoft Windows specification, Plane 0 is the 'XOR' mask; Plane 1 is the bitwise-inverse of the 'AND' mask.

Table 5-10. Hardware Cursor Planes

NOT–AND Mask Value (Plane 1 Mask)	XOR Mask Value (Plane 0 Mask)	Resulting Pixel Color on Display
0	0	Transparent
0	1	Inverted
1	0	Cursor color 0
1	1	Cursor color 1

The two hardware cursor colors are stored as two extra colors in the DAC. These two extra DAC entries are accessed by setting PALETTE_STATE[3]. The cursor colors are read or written by the standard VGA DAC registers (PALETTE_READ_ADDRESS, PALETTE_WRITE_ADDRESS, and PALETTE_DATA). Cursor color 0 is read or written at index 00h and cursor color 1 is read or written at index 0Fh.

When PALETTE_STATE[3] is set, neither the CPU nor the CL-GD5462 pixel path can access standard palette registers 0–255. Therefore, only set the cursor colors with the display disabled (this is only a problem in 256-color modes). Access the cursor color palette entries in the following sequence:

1) Disable the display.

2) Set PALETTE_STATE[3].

3) Read or write the cursor colors.

4) Clear PALETTE_STATE[3].

5) Enable the display.

The hardware cursor bit masks can be stored on any 1024-byte boundary within the first 8 Mbytes of display memory. The linear offset, from the start of the frame buffer to the hardware cursor storage area, is stored in the CURSOR_LOCATION register. Actually, only bits 22:10 of the linear offset are written to this register. The linear offset must be shifted 8 bits right, and written to bits 14:2. Since the lower 10 bits of the linear offset are not used, the hardware cursor bit masks must begin on a 1024-byte boundary.

The CURSOR_LOCATION register allows the application to select one of any cursor patterns loaded in display memory to become the active graphics cursor. Any cursor pattern loaded into off-screen display memory must begin on a 1024-byte boundary. Determine the value to be written to the CURSOR_LOCATION register when tiling of display memory is not enabled. To determine the linear offset when tiling is enabled, see Section 5.4.

The hardware cursor bit masks total 1024 bytes in length. Thus, two sets of bit masks can fit into a single tile. Due to the 1024-byte boundary limit, the bit mask storage must begin at either the

first Kbyte within a tile or the 1024th byte within a tile. xByteOffset must be zero and yByteOffset must be zero or 1024. This means x must be on a 128-byte boundary and y must be on a tile height ÷ 2 boundary.

The planes of the bitmasks are interleaved on qword (64-bit) boundaries, with each qword containing one scanline of one plane of the hardware cursor. Starting at the CURSOR_LOCATION register, the hardware cursor bitmasks are stored as Plane 0, Scanline 0; Plane 1, Scanline 0; Plane 0, Scanline 1; and so on.

The hardware cursor position is controlled by programming the CURSOR_X and CURSOR_Y registers. Both registers have 12 bits defined for positioning the hardware cursor. Therefore, the x and y coordinates can range from 0–4095, with '00' being the upper-left corner of the display.

The CURSOR_PRESET register contains two 7-bit fields called X_PRESET and Y_PRESET. These preset fields control the offset within the bit masks displayed at the CURSOR_X and CURSOR_Y position on the screen. Setting both the X_PRESET and Y_PRESET to '0' causes the upper-left corner of the hardware cursor to display at CURSOR_X,CURSOR_Y. Setting the X_PRESET and/or Y_PRESET to a non-zero value allows the cursor to appear to move off the left or upper edge of the display. In general, the cursor hot spot is programmed into the CURSOR_PRESET register. This causes the active position within the cursor to display at the CURSOR_X,CURSOR_Y position on the screen.

5.3.2 Hardware Cursor Programming Example

These functions address the hardware cursor code example below. The hardware cursor functions make the following assumptions:

- The code is intended for 32-bit protected-mode operation. (This simplifies the HWGCLoadBitmasks code so that the programmer can write anywhere in the frame buffer with a single selector and does not have to be limited to the 64 Kbytes of 16-bit code.)
- The CL-GD546X Memory-Mapped registers are enabled.
- The Memory-Mapped registers are accessible by a selector variable – LagunaRegs.
- The CL-GD546X frame buffer is accessible by a selector variable – LagunaFB.

A brief description of the hardware cursor functions is listed in Table 5-11.

Table 5-11. Hardware Cursor Functions

Harware Cursor Function	Description
HWGCEnable	Enable hardware cursor.
HWGCDisable	Disable hardware cursor.
HWGCGetColors	Read hardware cursor palette entries.
HWGCSetColors	Write hardware cursor palette entries.
HWGCGetLinearOffset	Read hardware cursor offset.
HWGCSetLinearOffset	Write hardware cursor offset.
HWGCGetXY	Read hardware cursor xy position.
HWGCSetXY	Write hardware cursor xy position.

Table 5-11. Hardware Cursor Functions *(cont.)*

Harware Cursor Function	Description
HWGCGetPreset	Read hardware cursor xy preset values.
HWGCSetPreset	Write hardware cursor xy preset values.
HWGCLoadBitmasks	Write hardware cursor bit masks to display memory.
HWGCConvertXYToLinearOffset	Converts a given xy coordinate to a linear offset for tiled modes.

```
;*****************************************************************
;*            ****************************************
;*            * Copyright (c) 1995, Cirrus Logic, Inc. *
;*            *            All Rights Reserved         *
;*            ****************************************
;* FILE:                 hwcursor.asm
;* DESCRIPTION:          Programming examples for CL-GD546X Technical
;*                       Reference Manual section describing the Hardware
;*                       Cursor
;* ASSEMBLE WITH: ml /c /W3 hwcursor.asm; for MASM 6.11
;*****************************************************************

.386p
;*---------------------- EQUATES -------------------------------*;

; CL-GD546X memory mapped register offsets
PALETTE_STATE                         EQU         0x0B0       ; PALETTE_STATE
                                                              ; register
CURSOR_X                              EQU         0x0E0       ; CURSOR_X
register
CURSOR_Y                              EQU         0x0E2       ; CURSOR_Y
register
CURSOR_PRESET                         EQU         0x0E4       ; CURSOR_PRESET
                                                              ; register
CURSOR_CONTROL                        EQU         0x0E6       ; CURSOR_CONTROL
                                                              ; register
CURSOR_LOCATION                       EQU         0x0E8       ;
CURSOR_LOCATION
                                                              ; register

; Bit fields in CURSOR_CONTROL register
CURSOR_ENABLE_BIT                     EQU         0x0001      ; bit 0 enables/
                                                              ; disables hw
cursor
```



```
; Bit field masks for CURSOR_X and CURSOR_Y registers
CURSOR_X_MASK                   EQU         0x0FFF      ; CURSOR_X bits
of
                                                        ; CURSOR_X reg
CURSOR_Y_MASK                   EQU         0x0FFF      ; CURSOR_Y bits
of
                                                        ; CURSOR_Y reg


; Bit field masks for CURSOR_PRESET register
X_PRESET_MASK                   EQU         0x7F00      ; X_PRESET bits
of
                                                        ; CURSOR_PRESET
Y_PRESET_MASK                   EQU         0x007F      ; Y_PRESET bits
of
                                                        ; CURSOR_PRESET


; CURSOR_LOCATION register info
CURSOR_LOC_MASK                 EQU         0x7FFC      ; valid bits of
                                                        ;
CURSOR_LOCATION
CURSOR_LOC_SHIFT                EQU         8           ; number of bits
to
                                                        ; shift a 32-bit
offset
                                                        ; to the  right
                                                        ; before writing
to
                                                        ;
CURSOR_LOCATION


; Bit fields in PALETTE_STATE register
ACCESS_CURSOR_COLORS_BIT         EQU        0x08        ; bit 3 enables
access
                                                        ; to cursor
palette
                                                        ; entries
; DAC I/O ports
PALETTE_READ_ADDR               EQU         0x03C7
PALETTE_WRITE_ADDR              EQU         0x03C8
PALETTE_DATA                    EQU         0x03C9


; hardware cursor color indices
CURSOR_COLOR_0_INDEX            EQU         0x00
CURSOR_COLOR_1_INDEX            EQU         0x0F


; hardware cursor sizes
```

```
CURSOR_WIDTH                    EQU         64
CURSOR_HEIGHT                   EQU         64
CURSOR_WIDTH_BYTES              EQU         64/8

; Tile size info
BYTES_PER_RDRAM_BANK            EQU         1024*1024
BYTES_PER_TILE                  EQU         2048

;*---------------------- STRUCTURES -----------------------------*;

; This CURSOR_SHAPE structure is basically the structure used by
; Microsoft Windows 3.1 display drivers (except all elements have
; been expanded to dwords from words)

CURSOR_SHAPE                    STRUC
csHotX                          DD          0
csHotY                          DD          0
csWidth                         DD          0
csHeight                        DD          0
csWidthBytes                    DD          0
csColor                         DD          0
             ; Array of bytes that contain the bits for the AND and XOR
             ; masks. The first csHeight * csWidthBytes bytes contain the
             ; bits for the AND mask and the next csHeight * csWidthBytes
             ; bytes contain the bits for the XOR mask.  (Space is
             ; reserved for two 64 bit x 64 bit masks
;csBits                         DB          2*64/8*64 DUP (?)
                                            ; not part of structure!
CURSOR_SHAPE                    ENDS
;*---------------------- DATA SEGMENT ---------------------------*;

_DATA   segment DWORD USE32 PUBLIC 'DATA'
EXTERN                  LagunaRegs:WORD         ; selector to access memory
mapped
                                                ; regs
EXTERN                  LagunaFB:WORD           ; selector to access frame
buffer
EXTERN                  TileWidth:DWORD         ; must be set to 128, 256 or
2048
EXTERN                  TileHeight:DWORD        ; must be set to  16,   8 or
1
EXTERN                  BankInterleave:DWORD    ; must be set to 1, 2 or 4
EXTERN                  TilesPerLine:DWORD
```



```
; Sample Microsoft Windows style CURSOR_SHAPE structures for the
; standard arrow and hour glass cursors. (These are 32x32 cursors)
; arrowCursor CURSOR_SHAPE    < 0h, 0h, 20h, 20h, 04h, 0101h>
                DB      0FCh, 0FFh, 0FFh, 0FFh  ; AND mask scanline  0
                DB      0F8h, 0FFh, 0FFh, 0FFh  ;          scanline  1
                DB      0F0h, 0FFh, 0FFh, 0FFh  ;          scanline  2
                DB      0E0h, 0FFh, 0FFh, 0FFh  ;          scanline  3
                DB      0C0h, 0FFh, 0FFh, 0FFh  ;          scanline  4
                DB      080h, 0FFh, 0FFh, 0FFh  ;          scanline  5
                DB      000h, 0FFh, 0FFh, 0FFh  ;          scanline  6
                DB      000h, 0FEh, 0FFh, 0FFh  ;          scanline  7
                DB      000h, 0FCh, 0FFh, 0FFh  ;          scanline  8
                DB      000h, 0F8h, 0FFh, 0FFh  ;          scanline  9
                DB      000h, 0F0h, 0FFh, 0FFh  ;          scanline 10
                DB      000h, 0FFh, 0FFh, 0FFh  ;          scanline 11
                DB      000h, 0FFh, 0FFh, 0FFh  ;          scanline 12
                DB      008h, 0FEh, 0FFh, 0FFh  ;          scanline 13
                DB      01Ch, 0FEh, 0FFh, 0FFh  ;          scanline 14
                DB      03Eh, 0FCh, 0FFh, 0FFh  ;          scanline 15
                DB      03Fh, 0FCh, 0FFh, 0FFh  ;          scanline 16
                DB      07Fh, 0F8h, 0FFh, 0FFh  ;          scanline 17
                DB      07Fh, 0F8h, 0FFh, 0FFh  ;          scanline 18
                DB      0FFh, 0FCh, 0FFh, 0FFh  ;          scanline 19
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 20
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 21
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 22
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 23
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 24
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 25
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 26
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 27
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 28
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 29
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 30
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline 31
                DB      000h, 000h, 000h, 000h  ; XOR mask scanline  0
                DB      002h, 000h, 000h, 000h  ;          scanline  1
                DB      006h, 000h, 000h, 000h  ;          scanline  2
                DB      00Eh, 000h, 000h, 000h  ;          scanline  3
                DB      01Eh, 000h, 000h, 000h  ;          scanline  4
                DB      03Eh, 000h, 000h, 000h  ;          scanline  5
```

```
                DB      07Eh, 000h, 000h, 000h  ;          scanline  6
                DB      0FEh, 000h, 000h, 000h  ;          scanline  7
                DB      0FEh, 001h, 000h, 000h  ;          scanline  8
                DB      0FEh, 003h, 000h, 000h  ;          scanline  9
                DB      07Eh, 000h, 000h, 000h  ;          scanline 10
                DB      06Eh, 000h, 000h, 000h  ;          scanline 11
                DB      066h, 000h, 000h, 000h  ;          scanline 12
                DB      0C2h, 000h, 000h, 000h  ;          scanline 13
                DB      0C0h, 000h, 000h, 000h  ;          scanline 14
                DB      080h, 001h, 000h, 000h  ;          scanline 15
                DB      080h, 001h, 000h, 000h  ;          scanline 16
                DB      000h, 003h, 000h, 000h  ;          scanline 17
                DB      000h, 003h, 000h, 000h  ;          scanline 18
                DB      000h, 000h, 000h, 000h  ;          scanline 19
                DB      000h, 000h, 000h, 000h  ;          scanline 20
                DB      000h, 000h, 000h, 000h  ;          scanline 21
                DB      000h, 000h, 000h, 000h  ;          scanline 22
                DB      000h, 000h, 000h, 000h  ;          scanline 23
                DB      000h, 000h, 000h, 000h  ;          scanline 24
                DB      000h, 000h, 000h, 000h  ;          scanline 25
                DB      000h, 000h, 000h, 000h  ;          scanline 26
                DB      000h, 000h, 000h, 000h  ;          scanline 27
                DB      000h, 000h, 000h, 000h  ;          scanline 28
                DB      000h, 000h, 000h, 000h  ;          scanline 29
                DB      000h, 000h, 000h, 000h  ;          scanline 30
                DB      000h, 000h, 000h, 000h  ;          scanline 31

hourGlassCursor CURSOR_SHAPE    <10h, 10h, 20h, 20h, 04h, 0101h>
                DB      0FFh, 0FFh, 0FFh, 0FFh  ; AND mask scanline  0
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline  1
                DB      0FFh, 0FFh, 0FFh, 0FFh  ;          scanline  2
                DB      0FFh, 001h, 000h, 0FFh  ;          scanline  3
                DB      0FFh, 001h, 000h, 0FFh  ;          scanline  4
                DB      0FFh, 001h, 000h, 0FFh  ;          scanline  5
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline  6
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline  7
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline  8
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline  9
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline 10
                DB      0FFh, 003h, 080h, 0FFh  ;          scanline 11
                DB      0FFh, 007h, 0C0h, 0FFh  ;          scanline 12
                DB      0FFh, 00Fh, 0E0h, 0FFh  ;          scanline 13
```



```
DB      0FFh, 01Fh, 0F0h, 0FFh  ;           scanline 14
DB      0FFh, 03Fh, 0F8h, 0FFh  ;           scanline 15
DB      0FFh, 03Fh, 0F8h, 0FFh  ;           scanline 16
DB      0FFh, 03Fh, 0F8h, 0FFh  ;           scanline 17
DB      0FFh, 01Fh, 0F0h, 0FFh  ;           scanline 18
DB      0FFh, 00Fh, 0E0h, 0FFh  ;           scanline 19
DB      0FFh, 007h, 0C0h, 0FFh  ;           scanline 20
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 21
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 22
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 23
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 24
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 25
DB      0FFh, 003h, 080h, 0FFh  ;           scanline 26
DB      0FFh, 001h, 000h, 0FFh  ;           scanline 27
DB      0FFh, 001h, 000h, 0FFh  ;           scanline 28
DB      0FFh, 001h, 000h, 0FFh  ;           scanline 29
DB      0FFh, 0FFh, 0FFh, 0FFh  ;           scanline 30
DB      0FFh, 0FFh, 0FFh, 0FFh  ;           scanline 31
DB      000h, 000h, 000h, 000h  ; XOR mask  scanline  0
DB      000h, 000h, 000h, 000h  ;           scanline  1
DB      000h, 000h, 000h, 000h  ;           scanline  2
DB      000h, 000h, 000h, 000h  ;           scanline  3
DB      000h, 000h, 000h, 000h  ;           scanline  4
DB      000h, 0F8h, 03Fh, 000h  ;           scanline  5
DB      000h, 000h, 000h, 000h  ;           scanline  6
DB      000h, 0F0h, 01Fh, 000h  ;           scanline  7
DB      000h, 0F0h, 01Fh, 000h  ;           scanline  8
DB      000h, 0F0h, 01Bh, 000h  ;           scanline  9
DB      000h, 050h, 015h, 000h  ;           scanline 10
DB      000h, 0B0h, 01Ah, 000h  ;           scanline 11
DB      000h, 060h, 00Dh, 000h  ;           scanline 12
DB      000h, 0C0h, 006h, 000h  ;           scanline 13
DB      000h, 080h, 003h, 000h  ;           scanline 14
DB      000h, 000h, 001h, 000h  ;           scanline 15
DB      000h, 000h, 001h, 000h  ;           scanline 16
DB      000h, 000h, 001h, 000h  ;           scanline 17
DB      000h, 080h, 003h, 000h  ;           scanline 18
DB      000h, 0C0h, 007h, 000h  ;           scanline 19
DB      000h, 0E0h, 00Eh, 000h  ;           scanline 20
DB      000h, 0F0h, 01Fh, 000h  ;           scanline 21
DB      000h, 0F0h, 01Eh, 000h  ;           scanline 22
DB      000h, 070h, 01Dh, 000h  ;           scanline 23
```

```
                DB      000h, 0B0h, 01Ah, 000h  ;          scanline 24
                DB      000h, 050h, 015h, 000h  ;          scanline 25
                DB      000h, 000h, 000h, 000h  ;          scanline 26
                DB      000h, 0F8h, 03Fh, 000h  ;          scanline 27
                DB      000h, 000h, 000h, 000h  ;          scanline 28
                DB      000h, 000h, 000h, 000h  ;          scanline 29
                DB      000h, 000h, 000h, 000h  ;          scanline 30
                DB      000h, 000h, 000h, 000h  ;          scanline 31

_DATA ends

;*--------------------- CODE SEGMENT ---------------------------*;

_TEXT                   segment DWORD USE32 PUBLIC 'CODE'
                        assume cs:_TEXT, ds:_DATA, es:_DATA

;*****************************************************************
; FUNCTION:              HWGCEnable
; ENTRY:                 Nothing
; EXIT:                  Nothing
; DESCRIPTION:           Enables the hardware cursor
;                        Preserves all registers used
;
;*****************************************************************
            align       4
            public      HWGCEnable
HWGCEnable              PROC

            push        es                                  ; save regs used
            ; set hw cursor enable bit in CURSOR_CONTROL reg
            mov         es, LagunaRegs
            or          WORD PTR es:[CURSOR_CONTROL], CURSOR_ENABLE_BIT

            pop         es                                  ; restore regs
used

            ret

HWGCEnable              ENDP
;*****************************************************************
;
; FUNCTION:              HWGCDisable
```



```
; ENTRY:                Nothing
; EXIT:                 Nothing
; DESCRIPTION:          Disables the hardware cursor
;                       Preserves all registers used
;
;*****************************************************************
            align       4
            public      HWGCDisable
HWGCDisable             PROC

            push        es                                  ; save regs used
            ; clear hw cursor enable bit in CURSOR_CONTROL reg
            mov         es, LagunaRegs
            and         WORD PTR es:[CURSOR_CONTROL], NOT CURSOR_ENABLE_BIT

            pop         es                                  ; restore regs
used
            ret
HWGCDisable             ENDP

;*****************************************************************
; FUNCTION:             HWGCGetColors
; ENTRY:                Nothing
; EXIT:                 ESI =       Cursor Color 0 (R in AL, G in AH, B in 3rd
;                                   byte of esi)
;                       EDI =       Cursor Color 1 (R in AL, G in AH, B in 3rd
;                                   byte of edi)
; DESCRIPTION:          Reads extra DAC entries for hardware cursor
;                       Preserves registers used except ESI and EDI
;
;*****************************************************************

            align       4
            public      HWGCGetColors
HWGCGetColors           PROC

            push        es                                  ; save regs used
            push        eax
            push        edx
            ; enable access to cursor colors
            mov         es, LagunaRegs
```

```
                or          BYTE PTR es:[PALETTE_STATE],
                                    ACCESS_CURSOR_COLORS_BIT

                ; set up to read cursor color 0
                mov         edx, PALETTE_READ_ADDR
                mov         al, CURSOR_COLOR_0_INDEX
                out         dx, al

                ; read cursor color 0
                mov         edx, PALETTE_DATA
                xor         eax, eax                        ; clear eax
                in          al, dx                          ; read red value
                ror         eax, 8                          ; put red in high
byte
                                                            ; of eax
                in          al, dx                          ; read green
value
                ror         eax, 8                          ; put green in
high byte
                                                            ; of eax
                in          al, dx                          ; read blue value
                ror         eax, 16                         ; put eax in
correct
                                                            ; format
                mov         esi, eax                        ; put Cursor
Color 0 in
                                                            ; esi

                ; set up to read cursor color 1
                mov         edx, PALETTE_READ_ADDR
                mov         al, CURSOR_COLOR_1_INDEX
                out         dx, al

                ; read cursor color 1
                mov         edx, PALETTE_DATA
                xor         eax, eax                        ; clear eax
                in          al, dx                          ; read red value
                ror         eax, 8                          ; put red in high
byte
                                                            ; of eax
                in          al, dx                          ; read green
value
                ror         eax, 8                          ; put green in
high byte
```



```
                                                        ; of eax
                in          al, dx                      ; read blue value
                ror         eax, 16                     ; put eax in
correct
                                                        ; format
                mov         edi, eax                    ; put Cursor
Color 1 in
                                                        ; edi

                ; disable access to cursor colors
                and         BYTE PTR es:[PALETTE_STATE],
                                        NOT ACCESS_CURSOR_COLORS_BIT

                pop         edx                         ; restore regs
used
                pop         eax
                pop         es

                ret

HWGCGetColors               ENDP

;*****************************************************************
;
; FUNCTION:                 HWGCSetColors
; ENTRY:                    ESI =       Cursor Color 0 (R in AL, G in AH, B in 3rd
;                                       byte of esi)
;                           EDI =       Cursor Color 1 (R in AL, G in AH, B in 3rd
;                                       byte of edi)
; EXIT:                     Nothing
; DESCRIPTION:              Writes hardware cursor colors to extra DAC entries
;                           Preserves registers used
;
;*****************************************************************

                align       4
                public      HWGCSetColors
HWGCSetColors               PROC

                push        es                          ; save regs used
                push        eax
                push        edx
```

```
                ; enable access to cursor colors
                mov         es, LagunaRegs
                or          BYTE PTR es:[PALETTE_STATE],
                                    ACCESS_CURSOR_COLORS_BIT

                ; set up to write cursor color 0
                mov         edx, PALETTE_WRITE_ADDR
                mov         al, CURSOR_COLOR_0_INDEX
                out         dx, al

                ; write cursor color 0
                mov         edx, PALETTE_DATA
                mov         eax, esi                            ; eax = Cursor
Color 0
                out         dx, al                              ; write red value
                shr         eax, 8
                out         dx, al                              ; write green
value
                shr         eax, 8
                out         dx, al                              ; write blue
value

                ; set up to write cursor color 1
                mov         edx, PALETTE_WRITE_ADDR
                mov         al, CURSOR_COLOR_1_INDEX
                out         dx, al

                ; write cursor color 1
                mov         edx, PALETTE_DATA
                mov         eax, edi                            ; eax = Cursor
Color 1
                out         dx, al                              ; write red value
                shr         eax, 8
                out         dx, al                              ; write green
value
                shr         eax, 8
                out         dx, al                              ; write blue
value

                ; disable access to cursor colors
                and         BYTE PTR es:[PALETTE_STATE],
                                    NOT ACCESS_CURSOR_COLORS_BIT
```



```
                pop         edx                                 ; restore regs
used
                pop         eax
                pop         es

                ret

HWGCSetColors               ENDP

;****************************************************************
;
; FUNCTION:                 HWGCGetLinearOffset
; ENTRY:                    Nothing
; EXIT:                     EAX =       32 bit offset in display memory where
;                                       hardware cursor bitmasks will be stored
; DESCRIPTION:              Reads offset of cursor bitmasks from CURSOR_LOCATION
;                           register. Preserves registers used except EAX
;
;****************************************************************

                align       4
                public      HWGCGetLinearOffset
HWGCGetLinearOffset                     PROC

                push        es                                  ; save regs used

                xor         eax, eax                            ; clear eax

                ; read CURSOR_LOCATION register
                mov         es, LagunaRegs
                mov         ax, WORD PTR es:[CURSOR_LOCATION]

                ; adjust to a 32-bit offset
                and         ax, CURSOR_LOC_MASK                 ; mask off unused
bits
                shl         eax, CURSOR_LOC_SHIFT               ; shift left to
                                                                ; appropriate 32-
bit
                                                                ; value

                pop         es                                  ; restore regs
used
```

```
                ret

HWGCGetLinearOffset                     ENDP

;****************************************************************
;
; FUNCTION:                 HWGCSetLinearOffset
; ENTRY:                    EAX =       32 bit offset in display memory where
;                                       hardware cursor bitmasks will be stored
; EXIT:                     Nothing
; DESCRIPTION:              Writes offset of cursor bitmasks to CURSOR_LOCATION
;                           register. Preserves registers used
;
;****************************************************************

                align       4
                public      HWGCSetLinearOffset
HWGCSetLinearOffset                     PROC

                push        es                                  ; save regs used
                push        eax

                ; adjust 32-bit offset to correct bits for CURSOR_LOCATION
                ; reg
                shr         eax, CURSOR_LOC_SHIFT
                and         ax, CURSOR_LOC_MASK                 ; ax = bits 22-10
of
                                                                ; offset
                ; write CURSOR_LOCATION reg
                mov         es, LagunaRegs
                mov         WORD PTR es:[CURSOR_LOCATION], ax

                pop         eax                                 ; restore regs
used
                pop         es

                ret

HWGCSetLinearOffsetENDP

;****************************************************************
;
```



```
; FUNCTION:             HWGCGetXY
; ENTRY:                Nothing
; EXIT:                 EAX =       current cursor position
;                                   CURSOR_Y highword, CURSOR_X loword
; DESCRIPTION:          Reads current cursor position from CURSOR_X and
;                       CURSOR_Y registers, reads both values simultaneously
;                       as a dword. Preserves registers used except EAX
;
;*****************************************************************

            align       4
            public      HWGCGetXY
HWGCGetXY               PROC

            push        es                                  ; save regs used

            ; read CURSOR_X and CURSOR_Y as a dword
            mov         es, LagunaRegs
            mov         eax, DWORD PTR es:[CURSOR_X]
            ; clear reserved bits
            and         eax, (CURSOR_Y_MASK SHL 16) OR CURSOR_X_MASK

            pop         es                                  ; restore regs
used

            ret

HWGCGetXY               ENDP

;*****************************************************************
;
; FUNCTION:             HWGCSetXY
; ENTRY:                EAX =       current cursor position
;                                   CURSOR_Y highword, CURSOR_X loword
; EXIT:                 Nothing
; DESCRIPTION:          Writes current cursor position to CURSOR_X and
;                       CURSOR_Y registers, writes both values
;                       simultaneously as a dword. Preserves registers used
;
;*****************************************************************

            align       4
```

```
                public      HWGCSetXY
HWGCSetXY                   PROC

                push        es                                  ; save regs used
                ; clear reserved bits
                and         eax, (CURSOR_Y_MASK SHL 16) OR CURSOR_X_MASK

                ; write CURSOR_X and CURSOR_Y as a dword
                mov         es, LagunaRegs
                mov         DWORD PTR es:[CURSOR_X], eax

                pop         es                                  ; restore regs
used

                ret

HWGCSetXY                   ENDP

;***************************************************************
;
; FUNCTION:                 HWGCGetPreset
; ENTRY:                    Nothing
; EXIT:                     AX =        current cursor preset
;                                       AL = Y_PRESET, AH = X_PRESET
; DESCRIPTION:              Reads current cursor presets from CURSOR_PRESET
;                           register. Preserves registers used except AX
;
;***************************************************************

                align       4
                public      HWGCGetPreset
HWGCGetPreset               PROC

                push        es                                  ; save regs used

                ; read CURSOR_PRESET reg
                mov         es, LagunaRegs
                mov         ax, WORD PTR es:[CURSOR_PRESET]

                ; clear reserved bits
                and         ax, X_PRESET_MASK OR Y_PRESET_MASK
```



```
                pop         es                                  ; restore regs
used

                ret

HWGCGetPreset               ENDP

;***************************************************************
;
; FUNCTION:                 HWGCSetPreset
; ENTRY:                    AX =        cursor preset
;                           AL =        Y_PRESET, AH = X_PRESET
; EXIT:                     Nothing
; DESCRIPTION:              Writes cursor preset values to CURSOR_PRESET
                            register. Preserves registers used
;
;***************************************************************

                align       4
                public      HWGCSetPreset
HWGCSetPreset               PROC

                push        es                                  ; save regs used

                ; clear reserved bits
                and         ax, X_PRESET_MASK OR Y_PRESET_MASK

                ; write CURSOR_PRESET reg
                mov         es, LagunaRegs
                mov         WORD PTR es:[CURSOR_PRESET], ax

                pop         es                                  ; restore regs
used

                ret

HWGCSetPreset               ENDP

;***************************************************************
;
; FUNCTION:                 HWGCLoadBitmasks
; ENTRY:                    ESI =       address of CURSOR_SHAPE structure
variable
```

```
;                       EDI =       offset into display memory where bit masks
;                                   are to be stored
; EXIT:                 Nothing
; DESCRIPTION:          Copies AND and XOR bitmasks from CURSOR_SHAPE
;                       structure to offscreen display memory.  Writes XOR
;                       bitmask to Plane 0 bitwise inverts AND mask and
;                       writes to Plane 1.  The cursor bitmasks are either
                        truncated or padded to 64x64 depending on the size
                        given in the CURSOR_SHAPE structure.Preserves
;                       registers used
;
;*************************************************************

numPaddingBytes                 EQU         DWORD PTR [ebp-20]
numPaddingScanLines             EQU         DWORD PTR [ebp-16]
andMaskOffset                   EQU         DWORD PTR [ebp-12]
xorMaskOffset                   EQU         DWORD PTR [ebp-8]
maskPitch                       EQU         DWORD PTR [ebp-4]

            align       4
            public      HWGCLoadBitmasks
HWGCLoadBitmasks                PROC

            push        ebp                             ; set up stack
frame
            mov         ebp, esp
            sub         esp, 20                         ; make room on
stack for
                                                        ; local vars
            push        es                              ; save regs used
            push        eax
            push        ebx
            push        ecx
            push        edx
            push        esi
            push        edi
            mov         es, LagunaFB                    ; es:edi -> frame
buffer

; set up local pointers to bitmasks and save copy of bitmask pitch

            mov         ecx, esi
            add         ecx, SIZE CURSOR_SHAPE
```



```
                mov         andMaskOffset, ecx

                mov         eax, [esi].CURSOR_SHAPE.csWidthBytes
                mov         maskPitch, eax
                mul         [esi].CURSOR_SHAPE.csHeight
                add         ecx, eax
                mov         xorMaskOffset, ecx

        ; determine number of bytes to pad in x direction
        ; if the passed in masks are wider than 64 bits then numPaddingBytes
        ; will be less than zero

                mov         eax, CURSOR_WIDTH_BYTES
                sub         eax, [esi].CURSOR_SHAPE.csWidthBytes
                mov         numPaddingBytes, eax

        ; determine number of scanlines to pad in y direction
        ; if the passed in masks are higher than 64 bits then
        ; numPaddingScanLines will be less than zero
                mov         eax, CURSOR_HEIGHT
                sub         eax, [esi].CURSOR_SHAPE.csHeight
                mov         numPaddingScanLines, eax

        ; Copy bitmasks to frame buffer.  ebx is set to number of scanlines
        ; contained in masks passed in or CURSOR_HEIGHT, whichever is
        ; smaller.  edx is set to the number of bytes per scanline contained
        ; in masks passed in or CURSOR_WIDTH_BYTES whichever is smaller.

                mov         ebx, [esi].CURSOR_SHAPE.csHeight
                cmp         ebx, CURSOR_HEIGHT
                jle         @f
                mov         ebx, CURSOR_HEIGHT
        @@:
                mov         edx, [esi].CURSOR_SHAPE.csWidthBytes
                cmp         edx, CURSOR_WIDTH_BYTES
                jle         @f
                mov         edx, CURSOR_WIDTH_BYTES
        @@:

        ; Loop over bitmask data passed in and copy to frame buffer,
        ; conversion of Microsoft Windows format cursor data to CL-GD546X
        ; format cursor data is the XOR mask is written to Plane 0 and the
```

```
; AND mask is bitwise inverted and written to Plane 1.  Recall that
; the frame buffer must be loaded as Plane 0, Scanline 0; Plane 1,
; Scanline 0; Plane 0, Scanline 1; etc.
; If padding is required the AND mask should be padded with 1's
; (meaning Plane 1 should be padded with 0's) and the XOR mask should
; be padded with 0's. Thus both CL-GD546X planes should be padded
; with 0's. Note that EDI is still pointing to the frame buffer.

NextScanLine:
             mov         esi, xorMaskOffset                ; point to XOR
mask
             mov         ecx, edx                          ; get number of
bytes in
                                                           ; scanline
             rep  movsb                                    ; copy XOR
scanline to
                                                           ; frame buffer
             mov         esi, maskPitch                    ; update XOR mask
                                                           ; pointer to
             add         xorMaskOffset, esi                ; next scanline

             mov         ecx, numPaddingBytes              ; see if any
padding
                                                           ; needed
             cmp         ecx, 0
             jle         NoXorPadding                      ; no, then skip
padding
                                                           ; code
             xor         al, al                            ; pad remainder
of
                                                           ; scanline with
0's
   rep  stosb

NoXorPadding:

             mov         esi, andMaskOffset                ; point to AND
mask
             mov         ecx, edx                          ; get number of
bytes in
                                                           ; scanline

NextAndByte:
```



```
                lodsb                                   ; read byte from
AND
                                                        ; mask
                not         al                          ; invert it
                stosb                                   ; write it to
frame
                                                        ; buffer
                dec         ecx                         ; are we done?
                jnz         NextAndByte                 ; no, go do next
byte in
                                                        ; scanline
                mov         ecx, numPaddingBytes        ; see if any
padding
                                                        ; needed
                cmp         ecx, 0
                jle         NoAndPadding                ; no, then skip
padding
                                                        ; code

                xor         al, al                      ; pad remainder
of
                                                        ; scanline with
0's
   rep  stosb

NoAndPadding:

                mov         esi, maskPitch
                add         andMaskOffset, esi
                dec         ebx                         ; have we done
all the
                                                        ; scanlines?
                jnz         NextScanLine                ; no, go do next
                                                        ; scanline
                mov         ecx, numPaddingScanLines
                cmp         ecx, 0                      ; see if we need
to
                                                        ; pad any
scanlines
                jle         NoScanLinePadding           ; no, then skip
padding
                                                        ; code

                ; determine number of dwords to fill with 0's for padding
                ; scanlines. 2 dwords per scanline per mask * numScanLines
```

```
                ; to pad
                imul        ecx, 2*(CURSOR_WIDTH_BYTES SHR 2)

                xor         eax, eax                            ; pad area with
0's
   rep  stosd

NoScanLinePadding:

                pop         edi                                 ; restore regs
used
                pop         esi
                pop         edx
                pop         ecx
                pop         ebx
                pop         eax
                pop         es

                mov         esp, ebp                            ; clear stack
frame
                pop         ebp

                ret

HWGCLoadBitmasks                    ENDP

;*****************************************************************
;
; FUNCTION:             HWGCConvertXYToLinearOffset
; ENTRY:                EAX =       x in bytes, (x-pixel * (bpp/8))
;                       ECX =       y
; EXIT:                 EAX =       linear offset
; DESCRIPTION:          Converts a given x,y coordinate to the linear offset
;                       from the start of the frame buffer in tiled modes.
;                       To support 8, 16 and 32 bpp with the same function,
;                       the x coordinate passed in eax must already be in
;                       bytes (that is the x pixel coordinate must be
;                       multiplied by the bytes per pixel).
;                       This function relies on a few external variables
;                       containing the tile configuration which must be set
;                       up prior to calling this function.  The varaibles
;                       are TileWidth, TileHeight, BankInterleave and
;                       TilesPerLine. Preserves registers used except EAX
```



```
;
;*****************************************************************

xTile                   EQU         DWORD PTR [ebp-16]
yTile                   EQU         DWORD PTR [ebp-12]
tileNo                  EQU         DWORD PTR [ebp-8]
byteOffset              EQU         DWORD PTR [ebp-4]

            align       4
            public      HWGCConvertXYToLinearOffset
HWGCConvertXYToLinearOffset                     PROC

            push        ebp                             ; set up stack
frame
            mov         ebp, esp
            sub         esp, 16                         ; make room on
stack for
                                                        ; local vars
            push        ebx                             ; save regs used
            push        ecx
            push        edx

; calculate x tile coordinate and byte offset into tile due to x
; xTile       = x / TileWidth
; xByteOffset = x % TileWidth

            xor         edx, edx
            mov         ebx, TileWidth
            div         ebx                             ; divide edx:eax
by
                                                        ; TileWidth
            mov         xTile, eax                      ; quotient is
xTile
            mov         byteOffset, edx                 ; remainder is
                                                        ; xByteOffset

; calculate y tile coordinate and byte offset into tile due to y
; yTile       =  y / TileHeight
; yByteOffset = (y % TileHeight) * TileWidth
; byteOffset  = yByteOffset + xByteOffset

            xor         edx, edx
            mov         eax, ecx
```

```
                div         TileHeight                          ; divide edx:eax
by
                                                                ; TileHeight
                mov         yTile, eax                          ; quotient is
yTile
                mov         eax, edx                            ; put remainder
in eax
                mul         ebx                                 ; multiply by
tileWidth
                add         byteOffset, eax                     ; add yByteOffset
to
                                                                ; xByteOffset

; calculate tile number from start of RDRAM bank
; tileNo = (yTile * TilesPerLine) / Interleave + xTile

                xor         edx, edx
                mov         ebx, BankInterleave
                mov         eax, yTile
                mul         TilesPerLine                        ; yTile *
TilesPerLine
                div         ebx                                 ; divide by
                                                                ; BankInterleave
                add         eax, xTile                          ; add xTile
                mov         tileNo, eax

; determine RDRAM bank number
; BankNo = ((xTile % Interleave) + (yTile % Interleave)) % Interleave

                xor         edx, edx
                mov         eax, xTile
                div         ebx                                 ; divide xTile by
                                                                ; BankInterleave
                mov         ecx, edx                            ; save remainder
in ecx

                xor         edx, edx
                mov         eax, yTile
                div         ebx                                 ; divide yTile by
                                                                ; BankInterleave
                add         ecx, edx                            ; add remainder
to ecx

                xor         edx, edx
```



```
                mov         eax, ecx
                div         ebx                             ; divide sum of
above
                                                            ; remainders by
                                                            ;
BankInterleave, this
                                                            ; leaves BankNo
in edx

; calculate linear offset
;   LinOffset = BankNo * BYTES_PER_RDRAM_BANK +
;               tileNo * BYTES_PER_TILE +
;               yByteOffset + xByteOffset

                mov         eax, edx                        ; put BankNo in
eax
                mov         ebx, BYTES_PER_RDRAM_BANK
                mul         ebx                             ; BankNo *
BYTES_PER_
                                                            ; RDRAM_BANK
                mov         ecx, eax                        ; save in ecx

                mov         eax, tileNo                     ; put tileNo in
eax
                mov         ebx, BYTES_PER_TILE
                mul         ebx                             ; tileNo *
                                                            ; BYTES_PER_TILE
                add         ecx, eax                        ; add to ecx

                add         ecx, byteOffset                 ; add
yByteOffset+
                                                            ; xByteOffset to
ecx

                mov         eax, ecx                        ; put in eax for
return
                                                            ; to caller


                pop         edx                             ; restore regs
used
                pop         ecx
                pop         ebx

                mov         esp, ebp                        ; clear stack
frame
```

```
                pop         ebp
                ret

HWGCConvertXYToLinearOffset                         ENDP

_TEXT           ends

end
```

5.4 Linear Offset in Tiled Mode

When the CL-GD546X is configured for Tiled mode, the linear address offset of a given xy coordinate of display memory is affected by the width and height of the tiles, and the interleave of RDRAM memory banks.

The following description assumes that tiling is enabled, and that the hardware cursor is stored at some given xy coordinate of off-screen display memory. The upper-left corner of display memory is assumed to be the upper-left corner of the screen (xy = 00) and the beginning of display memory (offset = 0). It is further assumed that the x coordinate is normalized to the width of a scanline. For example, if the display memory is configured for a pitch of 1024 pixels, then x is between 0 and 1023.

Tiles used with the CL-GD546X are 2048 bytes and can be configured to have a width of 128, 256, or 2048 bytes. The tiles then have a corresponding height of 16, 8, or 1 scanlines, respectively. For example, assume tiles are configured for 128 × 16 at 8 bpp without bank interleave. The first 128 bytes of display memory maps to the first 128 pixels of scanline 0. The next 128 bytes of display memory maps to the first 128 pixels of scanline 1. Therefore, the first 2048 bytes map to the first 128 pixels of scanlines 0–15. Bytes 2048–2175 of display memory map to pixels 128–255 on scanline 0, and so on. Mapping of display memory continues from left to right in 128-pixel blocks until the first 16 scanlines are mapped. For example, at 1024 × 768 with 8 bpp and 128 × 16 tiles, there are eight tiles across display memory when mapping the first 16 scanlines. Then, the mapping restarts at the left edge of the display for the second block of 16 scanlines. The addition of RDRAM bank interleave further complicates mapping.

Table 5-12 shows the layout of tiles in display memory at 1024 × 768. In the 'No Interleave' layout: Tile 0 is the first 2048 bytes of display memory; Tile 1 is the next 2048 bytes; and so on. Each number in the following tables represents a tile number. In the two- and four-way interleave layouts (Table 5-13 and Table 5-14), Tile '0,0' is the first 2048 bytes of display memory in bank 0; Tile '1,0' is the first 2048 bytes of display memory in bank 1; Tile '0,1' is the second 2048 bytes of display memory in bank 0; and so on.

NOTE: Each group of numbers in Table 5-13 and Table 5-14 represent first the bank number followed by a comma, then the tile number.

Table 5-12. 1024 × 768 × 8, 1-Mbyte RDRAM, No Interleave, 128 × 16 Tile Size

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Table 5-13. 1024 × 768 × 16, 2-Mbyte RDRAM, Two-way Interleave, 128 × 16 Tile Size

0,0	1,1	0,2	1,3	0,4	1,5	0,6	1,7
1,0	0,1	1,2	0,3	1,4	0,5	1,6	0,7
0,8	1,9	0,10	1,11	0,12	1,13	0,14	1,15
1,8	0,9	1,10	0,11	1,12	0,13	1,14	0,15
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Table 5-14. 1024 × 768 × 8, 4-Mbyte RDRAM, Four-way Interleave, 128 × 16 Tile Size

0,0	1,1	2,2	3,3	0,4	1,5	2,6	3,7
1,0	2,1	3,2	0,3	1,4	2,5	3,6	0,7
2,0	3,1	0,2	1,3	2,4	3,5	0,6	1,7
3,0	0,1	1,2	2,3	3,4	0,5	1,6	2,7
0,8	1,9	2,10	3,11	0,12	1,13	2,14	3,15
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

The frame buffer is an array of tiles. The width of the array of tiles is the TileScreenPitch, computed by using the Equation 5-11.

$$TileScreenPitch = \frac{ScreenPitchInBytes}{TileWidth}$$

Equation 5-11

The ScreenPitchInBytes must be an integer multiple of the TileWidth.

Given an xy pixel coordinate, the programmer can determine which xTile, yTile coordinate contains the xy pixel coordinate by using Equation 5-12 and Equation 5-13 (assuming integer math and truncation).

$$xTile = \left\langle x \bullet \frac{bpp}{8} \right\rangle \div (TileWidth)$$

Equation 5-12

yTile can be calculated using Equation 5-13.

$$yTile = \frac{y}{TileHeight}$$

Equation 5-13

where,
bpp is the bpp that are 8, 16, or 32. (This does not apply for 24-bpp Packed Pixel modes.)

If the interleave factor is ignored, the programmer can determine which tile contains the desired xy pixel coordinate by using Equation 5-14.

$$TileNo = \langle yTile \bullet TileScnPitch \rangle + xTile$$

Equation 5-14

The offset from the start of a given tile to the desired xy pixel coordinate is then the remainder of the x bytes, plus the remainder of the y scanlines times the TileWidth in bytes. So the x and y offsets in bytes from the start of a tile can be determined by using Equation 5-15 and Equation 5-16.

$$xByteOffset = \left(x \bullet \frac{bpp}{8}\right) mod TileWidth$$

Equation 5-15

$$yByteOffset = (ymodTileHeight) \bullet TileWidth$$

Equation 5-16

The linear offset is computed using Equation 5-17 (ignoring interleave).

$$LinOffset = (TileNo \bullet BytesPerTile) + (YByteOffset + xByteOffset)$$

Equation 5-17

where,
BytesPerTile is 2048.

The interleave adds a BankNo to the yTile coordinate. If the interleave is two, then the programmer gets two rows of tiles in the y direction, one in each bank of RDRAM. If the interleave is four, then the programmer gets four rows of tiles in the y direction, before getting back to the first bank. Therefore, the programmer divides the yTile by the TileScnPitch value above, by the interleave factor to get the correct TileNo.

$$TileNo = \frac{yTile \bullet TileScnPitch}{Interleave + xTile}$$

Equation 5-18

The programmer also needs to determine which RDRAM BankNo the TileNo resides.

The BankNo is dependent on both xTile, yTile, and the interleave factor. Reference the tiling diagrams located at the beginning of this section.

For two-way interleave, the tiles in memory produce the basic Table 5-15.

Table 5-15. Two-way Interleave

		A	
		0	1
B	0	0	1
	1	1	0

For four-way interleave, the tiles in memory produce the basic table shown in Table 5-16.

Table 5-16. Four-way Interleave

		A			
		0	1	2	3
B	0	0	1	2	3
	1	1	2	3	0
	2	2	3	0	1
	3	3	0	1	2

Obviously, for no interleave (or one-way interleave) the BankNo is always Bank 0. BankNo is (A + B) mod interleave, where A = xTile mod interleave and B = yTile mod interleave. Therefore, BankNo is computed by using Equation 5-19.

$$BankNo = \frac{xTilemodInterleave + yTilemodInterleave}{Interleave}$$

Equation 5-19

Fortunately, for an interleave of one, both numerator terms are zero so BankNo is always zero.

In general, the linear offset is calculated using Equation 5-20 (including interleave factor) and the following equations define the qualifiers.

Equation 5-20

$$BankNo = ((xTilemodInterleave) + (yTilemodInterleave))modInterleave$$

Equation 5-21

$$TileNo = \left\| \frac{yTile \bullet TileScnPitch}{Interleave} \right\| \bullet xTile$$

Equation 5-22

$$TileScnPitch = \frac{ScnPitchInBytes}{TileWidth}$$

Equation 5-23

$$xTile = \frac{x \bullet \frac{bpp}{8}}{TileWidth}$$

Equation 5-24

$$yTile = \frac{y}{TileHeight}$$

Equation 5-25

$$xByteOffset = \left(x \bullet \frac{bpp}{8}\right) modTileWidth$$

Equation 5-26

$$yByteOffset = (ymodTileHeight) \bullet TileWidth$$

Equation 5-27

where,
xy, bpp, TileWidth, TileHeight, Interleave, and *ScnPitchInBytes* must be known from configuration data.

BYTES_PER_RDRAM_BANK depends on the configuration. However, it is a minimum of 1 Mbyte, since RDRAM comes in 1-Mbyte blocks. Each bank must be the same size. BYTES_PER_TILE is 2048.

5.5 Palette

This section describes how to enable and manipulate the CL-GD546X palette. The palette can be used in 8-bit color lookup modes to map pixel values to RGB color values. In True Color modes, the palette can be bypassed, or it can contain a gamma correction map for each color.

5.5.1 Programming the 8-bpp LUT

The CL-GD546X VGA accelerator is capable of supporting 256-color Packed Pixel graphics modes. The CL-GD546X integrated palette DAC consists of three addressable arrays, each with 256 8-bit entries and three 8-bit DACs. In 256 color graphics modes, this array functions as a 256 CLUT for selecting RGB color value, which is sent to the DAC.

In 256-color graphics mode, 8 bpp – or one byte-per-pixel – of data is stored in video memory. This 8-bit value is a index into the CLUT in palette DAC, to select the RGB color values for that pixel.

The COLOR registers listed in Table 5-17 write and read the 256-color palette LUT RGB values.

Table 5-17. COLOR Registers 8-bpp LUT

Color Register	Extended I/O Port	Memory-Mapped I/O Port	Description
Pixel Mask	3C6h	A0h	The bits in this register form the pixel mask for the palette DAC. In 256 colors, this register is initialized to 0FFh.
Palette Address Read Mode	Write-only 3C5h	Write-only A4h	This register contains the 8-bit address used to access one of the 256-color lookup entries during a read operation.
Palette Address Write Mode	Write-only 3C8h	Write-only A8h	This register contains the 8-bit address used to access one of the 256-color lookup entries during a write operation.
Pixel Data	3C9h	ACh	This field is the pixel data to the palette DAC.
DAC State	Read-only 3C7h	Read-only A4h	Bits 1:0 These bits indicate if the Palette Address Read register or the Palette Address Write register was the last to be accessed. A ‘00’ value indicates a write operation in-progress; ‘11’ indicates a read operation in-progress.

The Palette Data register is an 8-bit wide register that writes pixel data to the palette DAC entries. Prior to writing to this register, the Palette Address Write register (or the Palette Address Read register) is written with the palette address. Then three bytes of data, corresponding to the RED, GREEN, and BLUE values, are written to the palette DAC register. The RGB values are transferred on the third write and the palette address (write or read) is automatically incremented to the next address. This is useful when initializing more then one palette LUT entry in the palette DAC.

The Cirrus Logic CL-GD546X VGA BIOS supports the function calls listed in Table 5-18 to read from or write to the palette DAC lookup registers.

Table 5-18. Palette DAC Lookup Function Calls

VGA BIOS Function INT10h	Input		Output Parameters	Register Effected
	Register	Parameter		
Set Individual Color register (RAMDAC)	AH	10h	None	None
	AL	10h		
	BX	Color register		
	DH	Red		
	CH	Green		
	CL	Blue		
Set block of Color registers	AH	10h	None	Selected Color register values in palette DAC.
	AL	12h		
	BX	Start Color register		
	CX	Count		
	ES:DX	Color values		
Read individual Color register (RAMDAC)	AH	10h	DH: Red CH: Green CL: Blue	None
	AL	15h		
	BX	Color register		
Read block of Color registers	AH	10h	None	ES:DX buffer fills with three times the number in the CX register.
	AL	17h		
	BX	Start Color register		
	CX	Count		
	ES:DX	Buffer		

5.5.2 Programmable 6- or 8-bit Palette Entries

The CL-GD546X accelerator supports 256-color extended video modes in the resolutions listed in Table 5-19.

Table 5-19. Resolutions Supported in 256-Color Modes

Video Mode Number	VESA® Mode Number	Screen Resolution	Comments
13h	13h	320 × 200	Standard VGA packed pixel mode.
5Eh	100	640 × 400	CL-GD546X 8-bpp extended mode.
5Fh	101	640 × 480	CL-GD546X 8-bpp extended mode.
5Ch	103	800 × 600	CL-GD546X 8-bpp extended mode.
60h	105	1024 × 768	CL-GD546X 8-bpp extended mode.
6Ch	106	1280 × 1024	CL-GD546X 8-bpp extended mode.
7Bh	–	1600 × 1200	CL-GD546X 8-bpp extended mode.

In 256-color 8-bpp extended modes, the CLUT entries can be configured for 6- or 8-bit Red, Green, and Blue (RGB) values:

- The standard 6-bit palette DAC configuration of 6:6:6 RGB CLUT values allows selection of 256 simultaneous color out of a 256K-color palette.
- The extended 8-bit palette DAC configuration of 8:8:8 RGB CLUT values allows selection of 256 simultaneous color out of a 16.8M-color palette.

The SR7 extended Sequencer Mode register (extended I/O port 3C4h/3C5h, index 7) bit 0 selects 8-bpp packed mode.

The Pixel/Video Format register (memory-mapped I/O, offset C0h), bit 8 and SR7[0] select between 6-bit palette entries and 8-bit palette entries. (Refer to the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*).

5.6 System-Level Considerations

The CL-GD546X functions as a VGA adapter, a VGA adapter and a high-resolution adapter, or a high-resolution adapter. The VGA adapter is enabled/disabled by a strapping option. This allows different system configurations and a few are explained in the following sections.

5.6.1 Using Other Display Adapters

The CL-GD546X only works as a high-resolution display controller. There is a strapping bit VGA (P18) that disables the VGA. All accesses occur through memory-mapped I/O or the linear frame buffer. High Resolution mode is determined by reading the SubClass register, which returns 80h in this case instead of 00h. BIOS and driver software are able to use this feature to disable VGA accesses.

5.6.2 Using Multiple CL-GD546Xs

The process for using multiple CL-GD546X devices in one system is the same as the process for using other display adapters, except that one of the CL-GD546X devices is the primary VGA. Enable the strapping option on the primary CL-GD546X and disable the secondary CL-GD546X. For the frame buffer and Memory-Mapped registers, each CL-GD546X is assigned to its own unique physical address space.

5.6.3 Monitor Identification

The CL-GD546X has a built-in I^2C port. Communication with DDC-compliant monitors and serial bus EEPROMs is possible by following the I^2C serial bus protocol. An explanation of the I^2C Bus Protocol is found in the Philips I^2C bus specification (*Data Handbook I^2C Peripherals for Microcontrollers,* January 1992).

5.6.3.1 BIOS Interface

The BIOS provides an interface to read the monitor EDID by VESA/VBE Sub-Function 15. (See Section 6.6.3 in Chapter 6.)

5.6.3.2 Functional Example

The I^2C register is divided into two parts: input and output. The programmer writes to the output portion of the register and compares it with a read back from the input portion.

Inside a DDC-compliant monitor is a serial EEPROM, which contains the 128 bytes of EDID data. There are three different types of read operations the programmer can perform on an EEPROM: current address read, random read, and sequential read. The type of read performed below is a random read. This is accomplished by performing a dummy write to set the current word address, followed by a 128-byte read. The two procedures listed below are high-level descriptions of a dummy write and a random read.

Dummy Write

1) **Start Condition** — high-to-low on SDO with SCO high.
2) **Device Address** — mandatory one-zero sequence for the first significant bits, followed by three address bits (0 in this case), then by a low to indicate a write operation.
3) **Current Word Address** — eight zeroes.
4) **Stop Condition** — low-to-high on SDO with SCO high.

Random Read

1) **Start Condition** — high-to-low on SDO with SCO high.
2) **Device Address** — mandatory one-zero sequence for the first significant bits, followed by three address bits (0 in this case), then by a high to indicate a read operation.
3) **Read a byte, get an Ack** (for number (of bytes – 1) desired) — accomplished by keeping SDO low during the ninth clock cycle (SCO high).
4) Read a byte.
5) **Stop Condition** — low-to-high on SDO with SCO high.

With the exception that three bits for the device address are different, reading and writing to another EEPROM on the serial bus is very similar. The BIOS provides an interface for reading/writing to an EEPROM by the function Set/Get EEPROM Byte (ah = 12h, bl = b4h). See the Set/Get EEPROM byte function in Section 6.7 of Chapter 6.

The following example code illustrates how to use the I^2C bus to communicate with a monitor.

```
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

// Bit definition for MemMapped I/O 280h
// 15 = SCI - Serial Clock In
// 14 = Don't Care
// 13 = Don't Care
// 12 = Don't Care
// 11 = Don't Care
// 10 = Don't Care
// 9 = Don't Care
// 8 = SDI - Serial Data In
// 7 = SCO - Serial Clock Out
// 6 = Don't Care
// 5 = Don't Care
// 4 = Don't Care
// 3 = Don't Care
// 2 = Don't Care
// 1 = Don't Care
// 0 = SDO - Serial Data Out

char buffer[128];
int          error = 0;

GetMemMap - return the Segment of the Memory Mapped Register
unsigned char inmemb(int offset)
{            // This function reads a byte from memory-mapped I/O.
             unsigned char far * pByte = MK_FP(GetMemMap(), offset);
             return *pByte;
}
unsigned char outmemb(int offset, unsigned char value)
{            // This function writes a byte to memory-mapped I/O.
             unsigned char far * pByte = MK_FP(GetMemMap(), offset);
             *pByte = value;
             return *pByte;
```

```
}

void scl(int bit)                    // This function sets the clock line.
{
             int         data;
             delay(1);
             data = inmemb (0x280);
             data = (data & 0x7F) | (bit << 7);
             outmemb (0x280, data);
             delay(1);
}
void sda(int bit)                    // This function sets the data line.
{
             int         data;
             delay(1);
             if (bit) bit = 1;
             data = inmemb (0x281) & FEh | bit;
             outmemb (0x280, data);
             delay(1);
}

read_scl()                           // This function reads the clock line.
{
             delay(1);
             return      (inmemb(0x281) >> 7);
}
read_sda()                           // This function reads the data line.
{
             delay(1);
             return (inmemb(0x281) & 0x1);
}
void start(void)                     // This function is used to generate a
start
                                     //condition.
{
             scl(1);
             delay(1);
             sda(0);
             delay(1);
             scl(0);
             delay(1);
}
```



```
void stop(void)                         // This function is used to generate a
stop
                                        // condition.
{
            delay(1);
            scl(1);
            delay(1);
            sda(1);
}

int send_byte (unsigned char data)              // This function is used
                                                // to send a byte.
{
            int         i;
            for (i = 0; i < 8; i++)
            {
            delay(1);
                        sda((data << i) & 0x80);
            delay(1);
                        scl(1);
            delay(1);
                        while(read_scl() == 0)
                        {
                        printf ("nc\n");
                        };
            delay(1);
                        scl(0);
            }
            // Check for ACK
            sda(1);
            scl(1);
            delay(1);
            if (read_sda() == 1)
            {
                        printf ("no ACK\n");
                        return (1);
            }
            scl(0);
            return (0);
}
```

```
unsigned char read_byte (int ack)                // This function is used
                                                 // to read a byte.
{
            int         i, data;
            for (i = 0; i < 8; i++)
            {
            delay(1);
                        scl(1);
            delay(1);
                        data <<= 1;
                        data |= read_sda();
                        scl(0);
            delay(1);
            }
            delay(1);
            if (ack)
                        sda(0);
            else
                        sda(1);

            delay(1);
            scl(1);
            delay(1);
            scl(0);
            sda(1);
            return (data);
}

void main(void)                                  // This function is used to
read
                                                 // data from a monitor.
{
            int         i;

            // Dummy Write
            start();
            if (send_byte (0xA0))
            {
                        printf ("could not send command 'A0'\n");
            }
            if (send_byte (0x00))
            {
```



```
                printf ("could not send data '00'\n");
        }
        // Random Read
        start();
        if (send_byte (0xA1))
        {
                printf ("could not send command 'A1'\n");
        }
        for (i = 0; i < 127; i++)                       // Read 127 bytes
and
                                                        // get ack.
        {
                delay(1);
                buffer[i] = read_byte(1);
                if (error)
                {
                        printf ("could not read data\n");
                        break;
                }
        }
        buffer[i] = read_byte(0);                       // Get last and
send
                                                        // stop.
        stop();
        printf ("EDID table =\n");
        for (i = 0; i < sizeof (buffer); i++)
        {
                printf("%.2X  ", (unsigned char) buffer[i]);
        }
}
```

