User Manual for the Atari ST
Bit-Block Transfer Processor (BLiTTER)

The Atari Corporation
Sunnyvale, California
17 June 1987

Table of Contents

1. Introduction 2

2. Bit-Block Transfers 4

3. Functional Description 6

4. Programming Model 8
4.1. Register Map 8
4.2. Bit-Block Addresses 9
4.3. Bit-Block Alignments 11
4.4. Logic Operations 12
4.5. Halftone Operations 13
4.6. Bus Accesses 14

Appendix A -- Programming Example 15

Appendix B -- Extended ST BIOS Blit Configuration 23

Appendix C -- References 25

```
$Header: blitter.me,v 2.2 87/06/17 14:50:06 art Exp $

$Source: /u/art/0_docs/RCS/blitter.me,v $
$Author: art $

$Revision: 2.2 $
$Date: 87/06/17 14:50:06 $
$State: Exp $
$Locker: art $
$Log:   blitter.me,v $
#  Revision 2.2  87/06/17  14:50:06  art
#  Added new appendix for extended ST BIOS call.
#
#  Revision 2.1  87/06/15  15:10:54  art
#  Released.
#
#  Revision 1.3  87/06/15  14:58:36  art
#  Changed Bit-Block Transfers section after internal review.
#
#  Revision 1.2  87/06/12  02:28:39  art
#  Cleaned up previous hack.
#
#  Revision 1.1  87/06/02  13:58:33  art
#  Initial revision
#
```

THE SCOPE OF THIS DOCUMENT is limited to a functional description of the Atari ST BLiTTER. This document is not a data sheet for system integration, rather it is a user manual for system programming. For more information, please refer to the texts listed at the end of this document.

1. Introduction

The Atari ST Bit-Block Transfer Processor (BLiTTER) is a hardware implementation of the bit-block transfer (BitBlt aka blit) algorithm. BitBlt can be simply described as a procedure that moves bit-aligned data from a source location to a destination location through a given logic operation. The BitBlt primitive can be used to perform such operations as:

- Area seed filling
- Rotation by recursive subdivision
- Slice and smear magnification
- Brush line drawing using Bresenham DDA
- Text transformations eg bold, italic, outline
- Text scrolling
- Window updating
- Pattern filling

And general memory-to-memory block copying [1].

The heart of BitBlt was first formally defined by Newman and Sproull in their description of the function RasterOp [2]. As defined, RasterOp performed its block transfers on a bit-by-bit basis and was limited to a small subset of possible source and destination Boolean combinations. Enhancements to RasterOp such as processing bits in parallel or introducing a halftone pattern into the transfer were literally left as exercises for the reader.

In an effort to improve the functionality and performance of the original algorithm, the prescribed enhancements were incorporated into the definition of RasterOp and implemented in hardware as the RasterOp Chip [3]. However the RasterOp Chip lacked the two-dimensionality of the original function and suffered from a performance bottleneck caused by the loading and reloading of source, destination, and halftone data (ie it could not DMA).

While efforts were being made to improve the performance of RasterOp, the formal definition of RasterOp was further refined and became the basis of the BitBlt copyLoop primitive in the Smalltalk-80 graphics kernel [4]. Because of its comprehensive interface definition, the BitBlt

primitive was inefficient and required special-case optimizations that violated its general-purpose nature. Clearly a hardware solution was necessary to increase the performance of the BitBlt copyLoop without sacrificing its functionality.

The Atari ST BLiTTER is a hardware solution to the performance problems of BitBlt. The BLiTTER is a DMA device that implements the full BitBlt copyLoop definition with the addition of a few minor extensions. Single word or multiword increments and decrements are provided for transfers to destinations in Atari ST video display memory. A center mask, which would otherwise be a constant all ones, is also provided for an additional level of texture. The remainder of this document is directly based on the original functional description of the Atari ST BLiTTER.

2. Bit-Block Transfers

As previously stated, a bit-block transfer can be described as a procedure that moves bit-aligned data from a source location to a destination location through a given logic operation. There are sixteen logic combination rules associated with the merging of source and destination data. Note that this set contains all possible combinations between source and destination. The following table contains the valid BitBlt combination rules:

LOGIC OPERATIONS

(~s&~d) MSB	(~s&d)	(s&~d)	(s&d) LSB	OP	COMBINATION RULE
0	0	0	0	0	all zeros
0	0	0	1	1	source AND destination
0	0	1	0	2	source AND NOT destination
0	0	1	1	3	source
0	1	0	0	4	NOT source AND destination
0	1	0	1	5	destination
0	1	1	0	6	source XOR destination
0	1	1	1	7	source OR destination
1	0	0	0	8	NOT source AND NOT destination
1	0	0	1	9	NOT source XOR destination
1	0	1	0	A	NOT destination
1	0	1	1	B	source OR NOT destination
1	1	0	0	C	NOT source
1	1	0	1	D	NOT source OR destination
1	1	1	0	E	NOT source OR NOT destination
1	1	1	1	F	all ones

Adjustments to block extents and several other transfer parameters are determined prior to the invocation of the actual block transfer. These adjustments and parameters include clipping, skew, end masks, and overlap.

Clipping. The source and destination block extents are adjusted to conform with a specified clipping rectangle. Since both source and destination blocks are of equal dimension, the destination block extent is clipped to the extent of the source block (or vice versa). Note that the block transfer need not be performed if the resultant extent is zero.

Skew. The source-to-destination horizontal bit skew is calculated.

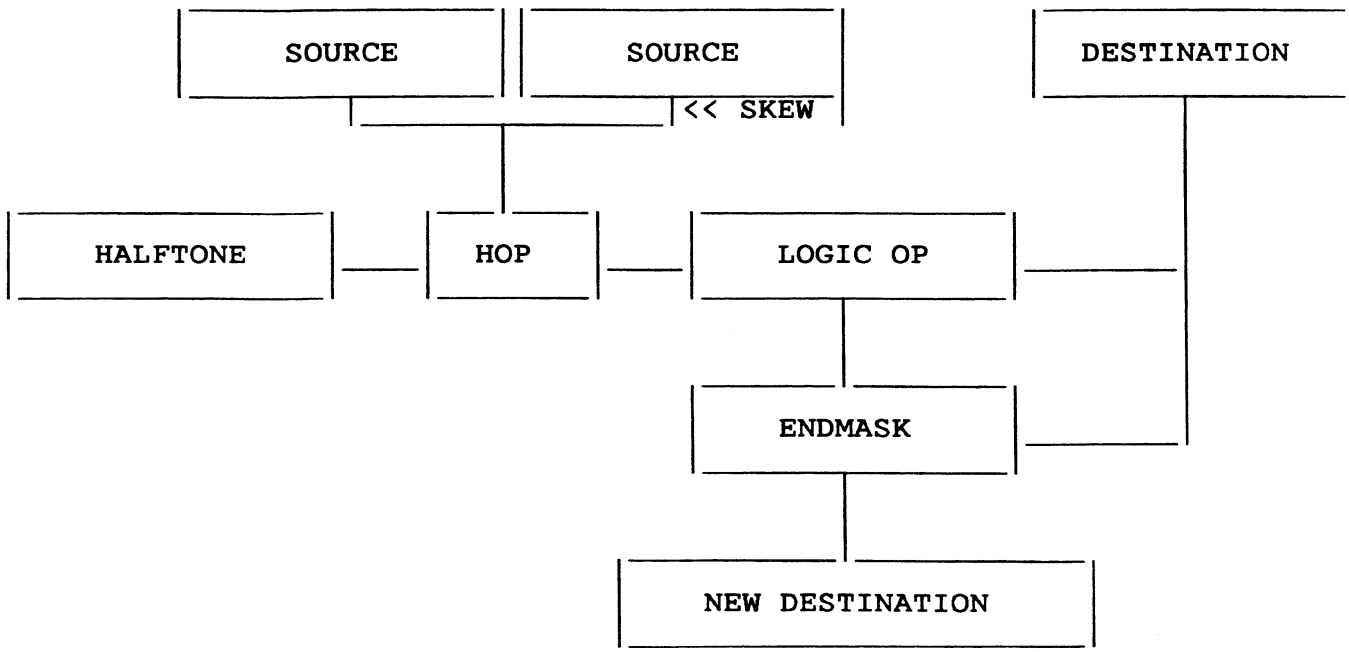
End Masks. The left and right partial word masks are determined. The masks are merged if the destination is one word in width.

Overlap. The block locations are checked for possible overlap in order to avoid the destruction of source data before it is transferred.

In non-overlapping transfers the source block scanning direction is inconsequential and can by default be from upper left to lower right. In overlapping transfers the source scanning direction is also from upper left to lower right if the source-to-destination transfer direction is up and/or to the left (ie source address is greater than or equal to destination address). However, if the overlapping source-to-destination transfer direction is down and/or to the right (ie source address is less than destination address), then the source data is scanned from lower right to upper left.

After the transfer parameters are determined the bit-block transfer operation can be invoked, transferring source to destination through the logic operation (HALFTONE and HOP will be described in the next section):

BIT-BLOCK TRANSFER



3. Functional Description

Please refer to the bit-block transfer diagram in the previous section. To understand how the components of a block transfer work, let's look at the simplest possible transfer. Take the case where we wish to fill a block of memory with either all zeros or all ones (OP = 0 or OP = F). In this case only the LOGIC OP block, which generates the ones or zeros, and the ENDMASK block are in the data path. If the end mask contains all ones, the BLiTTER will simply write one word after the other to the destination address without ever reading the destination.

As the writes take place the destination address will be adjusted according to the values in the DESTINATION X INCREMENT, DESTINATION Y INCREMENT, X COUNT, and Y COUNT registers. These registers define the size and shape of the block to be transferred. The X and Y COUNT registers define the size of the block. The X COUNT register specifies the number of word-size writes required to update one line of the destination. The Y COUNT register specifies the number of these lines in the block. The DESTINATION X INCREMENT register is a signed (2's complement) 16-bit quantity which is added to the destination address to calculate the address of the next destination word of the line. On the last write of the line the DESTINATION Y INCREMENT is added to calculate the address of the first word of the next line.

The end mask determines which bits of the destination word will be updated. Bits of the destination which correspond to ones in the end mask will be updated. Bits of the destination which correspond to zeros in the end mask will remain unchanged. Note that if any bits of the destination are to be left unchanged, a read-modify-write is required. In order to improve performance a read will only be performed if it is required. There are three ENDMASK registers numbered 1 through 3. ENDMASK 1 is used only for the first write of the line. ENDMASK 3 is used only for the last write of the line. ENDMASK 2 is used for all other writes.

Now let's consider a more complicated case, suppose we want to XOR a destination block with a 16x16 halftone pattern. First we load the HALFTONE RAM with the halftone pattern. Select halftone only using the HOP register and select source XOR destination using the OP register. The LINE NUMBER register is used to specify which of the 16 words of HALFTONE RAM is used for the current line. This register will be incremented or decremented at the end of each line according to the sign of the DESTINATION Y INCREMENT register. Set the DESTINATION X and Y INCREMENT and X and Y COUNT registers to the appropriate values and start the transfer. This same procedure can be followed to do the

combination using any logic operation by simply changing the value in the OP register. Similarly the combination can be performed using a source block instead of the HALFTONE RAM or using the logical AND of a source block and the HALFTONE RAM by changing the value of the HOP register. A source block is the same size as the destination block but may have different increments and address defined by the SOURCE X and Y INCREMENT and SOURCE ADDRESS registers.

Finally, let's look at the case when the source and destination blocks are not bit-aligned. In this case we may need to read the first two source words into the 32-bit source buffer and use the 16 bits that line up with the appropriate bits of the destination, as specified by the SKEW register. When the next source word is read, the lower 16 bits of the source buffer is transferred to the upper 16 bits and the lower is replaced by the new data. This process is reversed when the source is being read from the right to the left (SOURCE X INCREMENT negative). Since there are cases when it may be necessary for an extra source read to be performed at the beginning of each line to "prime" the source buffer and cases when it may not be necessary due to the choice of end mask, a bit has been provided which forces the extra read. The FXSR (aka pre-fetch) bit in the SKEW register indicates, when set, that an extra source read should be performed at the beginning of each line to "prime" the source buffer. Similarly the NFSR (aka post-flush) bit, when set, will prevent the last source read of the line. This read may not be necessary with certain combinations of end masks and skews. If the read is suppressed, the lower to upper half buffer transfer still occurs. Also in this case, a read-modify-write cycle is performed on the destination for the last write of each line regardless of the value of the corresponding ENDMASK register.

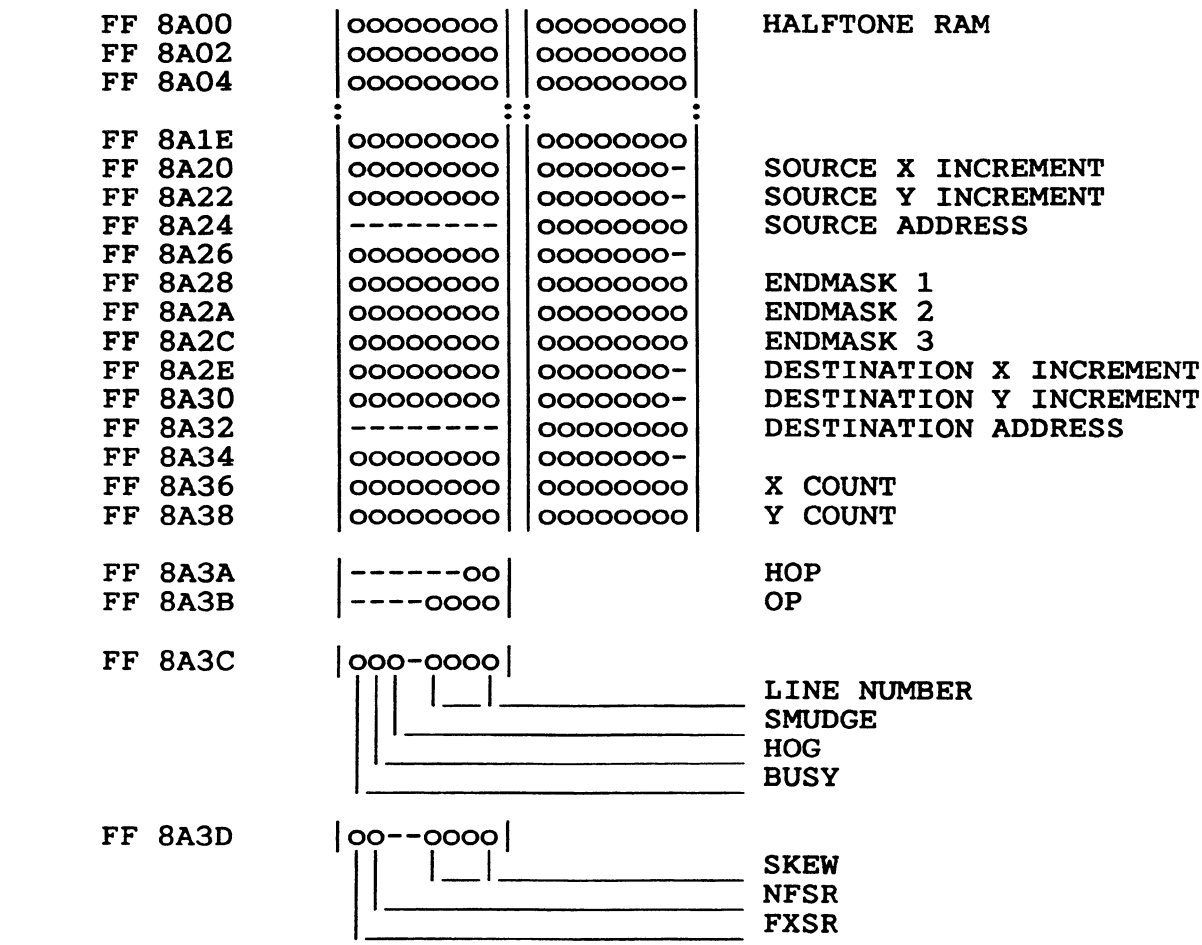
4. Programming Model

The BLiTTER contains a set of registers that specify bit-block addresses, bit-block alignments, logic and halftone operations, and bus accesses. The register set-up time remains practically constant and is large relative to small block transfers, whereas large bit-blocks are dominated by the execution time of the transfer itself.

4.1. Register Map

The following is a map of the BLiTTER programmable registers (note that all unused bits read back as zeros):

REGISTER MAP



4.2. Bit-Block Addresses

This subsection describes registers that specify bit-block origins, address increments, and extents.

SOURCE ADDRESS
: This 23-bit register contains the current address of the source field (only word addresses may be specified). It may be accessed using either word or long-word instructions. The value read back is always the address of the next word to be used in a source operation. It will be updated by the amounts specified in the SOURCE X INCREMENT and the SOURCE Y INCREMENT registers as the transfer progresses.

SOURCE X INCREMENT
: This is a signed 15-bit register, the least significant bit is ignored, specifying the offset in bytes to the address of the next source word in the current line. This value will be sign-extended and added to the SOURCE ADDRESS register at the end of a source word fetch, whenever the X COUNT register does not contain a value of one. If the X COUNT register is loaded with a value of one this register is not used. Byte instructions can not be used to read or write this register.

SOURCE Y INCREMENT
: This is a signed 15-bit register, the least significant bit is ignored, specifying the offset in bytes to the address of the first source word in the next line. This value will be sign-extended and added to the SOURCE ADDRESS register at the end of the last source word fetch of each line (when the X COUNT register contains a value of one). If the X COUNT register is loaded with a value of one this register is used exclusively. Byte instructions can not be used to read or write this register.

DESTINATION ADDRESS
: This 23-bit register contains the current address of the destination field (only word addresses may be specified). It may be accessed using either word or long-word instructions. The value read back is always the address of the next word to be modified in the destination field. It will be updated by the amounts specified in the DESTINATION X INCREMENT and the DESTINATION Y INCREMENT registers as the transfer progresses.

DESTINATION X INCREMENT
: This is a signed 15-bit register, the least significant bit is ignored, specifying the offset in bytes to the address of the next destination word in the current

line. This value will be sign-extended and added to the DESTINATION ADDRESS register at the end of a destination word write, whenever the X COUNT register does not contain a value of one. If the X COUNT register is loaded with a value of one this register is not used. Byte instructions can not be used to read or write this register.

DESTINATION Y INCREMENT

This is a signed 15-bit register, the least significant bit is ignored, specifying the offset in bytes to the address of the first destination word in the next line. This value will be sign-extended and added to the DESTINATION ADDRESS register at the end of the last destination word write of each line (when the X COUNT register contains a value of one). If the X COUNT register is loaded with a value of one this register is used exclusively. Byte instructions can not be used to read or write this register.

X COUNT

This 16-bit register specifies the number of words contained in one destination line. The minimum number is one and the maximum is 65536 designated by zero. Byte instructions can not be used to read or write this register. Reading this register returns the number of destination words yet to be written in the current line, NOT necessarily the value initially written to the register. Each time a destination word is written the value will be decremented until it reaches zero, at which time it will be returned to its initial value.

Y COUNT

This 16-bit register specifies the number of lines in the destination field. The minimum number is one and the maximum is 65536 designated by zero. Byte instructions can not be used to read or write this register. Reading this register returns the number of destination lines yet to be written, NOT necessarily the value initially written to the register. Each time a destination line is completed the value will be decremented until it reaches zero, at which time the tranfer is complete.

4.3. Bit-Block Alignments

This subsection describes registers that specify bit-block end masks, source-to-destination skew, and source data fetching.

ENDMASK 1, 2, 3

These 16-bit registers are used to mask destination writes. Bits of the destination word which correspond to ones in the current ENDMASK register will be modified. Bits of the destination word which correspond to zeros in the current ENDMASK register will remain unchanged. The current ENDMASK register is determined by position in the line. ENDMASK 1 is used only for the first write of a line. ENDMASK 3 is used only for the last write of a line. ENDMASK 2 is used in all other cases. In the case of a one word line ENDMASK 1 is used. Byte instructions can not be used to read or write these registers.

SKEW

The least significant four bits of the byte-wide register at FF 8A3D specify the source skew. This is the amount the data in the source data latch is shifted right before being combined with the halftone mask and destination data.

FXSR

FXSR stands for Force eXtra Source Read. When this bit is set one extra source read is performed at the start of each line to initialize the remainder portion source data latch.

NFSR

NFSR stands for No Final Source Read. When this bit is set the last source read of each line is not performed. Note that use of this and/or the FXSR bit the requires an adjustment to the SOURCE Y INCREMENT and SOURCE ADDRESS registers.

4.4. Logic Operations

This subsection describes registers that specify the logic combinations of source and destination bit-block data.

OP

The least significant four bits of the byte-wide register at FF 8A3B specify the source/destination combination rule according to the following table:

LOGIC OPERATIONS

OP	COMBINATION RULE
0	all zeros
1	source AND destination
2	source AND NOT destination
3	source
4	NOT source AND destination
5	destination
6	source XOR destination
7	source OR destination
8	NOT source AND NOT destination
9	NOT source XOR destination
A	NOT destination
B	source OR NOT destination
C	NOT source
D	NOT source OR destination
E	NOT source OR NOT destination
F	all ones

4.5. Halftone Operations

This subsection describes registers that specify the halftone pattern memory, halftone word index, and combinations of source and halftone data.

HALFTONE RAM
: This RAM holds a 16x16 halftone pattern mask. Each word is valid for one line of the destination field and is repeated every 16 lines. The current word is pointed to by the value in the LINE NUMBER register. These registers may be read, but can not be accessed using byte-wide instructions.

LINE NUMBER
: The least significant four bits of the byte-wide register at FF 8A3C specify the current halftone mask. The current value times two plus FF8A00 gives the address of the current halftone mask. This value is incremented or decremented at the end of each line and will wrap through zero. The sign of the DESTINATION Y INCREMENT determines if the line number is incremented or decremented (increment if positive, decrement if negative).

SMUDGE
: The SMUDGE bit, when set, causes the least significant four bits of the skewed source data to be used as the address of the current halftone pattern. Note that the halftone operation is still valid when SMUDGE is set.

HOP
: The least significant two bits of the byte-wide register at FF 8A3A specify the source/halftone combination rule according to the following table:

HALFTONE OPERATIONS

HOP	COMBINATION RULE
0	all ones
1	halftone
2	source
3	source AND halftone

4.6. Bus Accesses

This subsection describes registers that specify bus access control and BLiTTER start/status.

HOG

The HOG bit, when cleared, causes the processor and the BLiTTER to share the bus equally. In this mode each will get 64 bus cycles while the other is halted. When set, the bit will cause the processor to be halted until the transfer is complete. In either case the BLiTTER will yield to other DMA devices. Bus arbitration may allow the processor to execute one or more instructions even in hog mode. Therefore, don't assume that the instruction following the one which sets the BUSY bit will be executed only after the transfer is complete. The BUSY bit may be polled to achieve this kind of synchronization.

BUSY

The BUSY bit is set after all the other registers have been initialized to begin the transfer operation. It will remain set until the transfer is complete. The interrupt line is a duplicate of this bit. See the Programming Example for more details on how to use the BUSY bit.

Appendix A -- Programming Example

In order to maintain software compatibility with new or upgraded Atari STs equipped with the BLiTTER, software developers need only follow guidelines set forth by the VDI and "LINE A" documents. Revised TOS ROMs will work in concert with the BLiTTER, enhancing the performance of many VDI and "LINE A" operations. This occurs in a manner transparent to an executing program. Thus no special actions need be taken to utilize the performance advantages of the BLiTTER.

As a rule of thumb, never make a VDI or "LINE A" call from within an interrupt context since unpredictable and potentially catastrophic results will occur should one BLiTTER operation interrupt another BLiTTER operation.

The following program has not been optimized and is presented here for exemplary purposes only.

```
* (c) 1987 Atari Corporation
*     All Rights Reserved.

* BLiTTER BASE ADDRESS

BLiTTER         equ     $FF8A00

* BLiTTER REGISTER OFFSETS

Halftone        equ     0
Src_Xinc        equ     32
Src_Yinc        equ     34
Src_Addr        equ     36
Endmask1        equ     40
Endmask2        equ     42
Endmask3        equ     44
Dst_Xinc        equ     46
Dst_Yinc        equ     48
Dst_Addr        equ     50
X_Count         equ     54
Y_Count         equ     56
HOP             equ     58
OP              equ     59
Line_Num        equ     60
Skew            equ     61
```

```
* BLiTTER REGISTER FLAGS

fHOP_Source     equ     1
fHOP_Halftone   equ     0

fSkewFXSR       equ     7
fSkewNFSR       equ     6

fLineBusy       equ     7
fLineHog        equ     6
fLineSmudge     equ     5

* BLiTTER REGISTER MASKS

mHOP_Source     equ     $02
mHOP_Halftone   equ     $01

mSkewFXSR       equ     $80
mSkewNFSR       equ     $40

mLineBusy       equ     $80
mLineHog        equ     $40
mLineSmudge     equ     $20

*                   E n D m A s K   d A t A
*
* These tables are referenced by PC relative instructions.  Thus,
* the labels on these tables must remain within 128 bytes of the
* referencing instructions forever.  Amen.
*
* 0: Destination  1: Source   <<< Invert right end mask data >>>

lf_endmask:
        dc.w    $FFFF

rt_endmask:
        dc.w    $7FFF
        dc.w    $3FFF
        dc.w    $1FFF
        dc.w    $0FFF
        dc.w    $07FF
        dc.w    $03FF
        dc.w    $01FF
        dc.w    $00FF
        dc.w    $007F
        dc.w    $003F
        dc.w    $001F
        dc.w    $000F
        dc.w    $0007
        dc.w    $0003
        dc.w    $0001
        dc.w    $0000
```

```
* TiTLE:    BLiT_iT
*
* PuRPoSE:  Transfer a rectangular block of pixels located at an
*           arbitrary X,Y position in the source memory form to
*           another arbitrary X,Y position in the destination memory
*           form using replace mode (boolean operator 3).
*           The source and destination rectangles should not overlap.
*
* iN:
*       a4      pointer to 34 byte input parameter block
*
* NoTe:  This routine must be executed in supervisor mode as access
*        is made to hardware registers in the protected region of the
*        memory map.
*
*
*       I n P u T   p A r A m E t E r   B l O c K   o F f S e T s

SRC_FORM        equ      0 ; Base address of source memory form        .l
SRC_NXWD        equ      4 ; Offset between words in source plane      .w
SRC_NXLN        equ      6 ; Source form width                         .w
SRC_NXPL        equ      8 ; Offset between source planes              .w
SRC_XMIN        equ     10 ; Source blit rectangle minimum X           .w
SRC_YMIN        equ     12 ; Source blit rectangle minimum Y           .w

DST_FORM        equ     14 ; Base address of destination memory form   .l
DST_NXWD        equ     18 ; Offset between words in destination plane .w
DST_NXLN        equ     20 ; Destination form width                    .w
DST_NXPL        equ     22 ; Offset between destination planes         .w
DST_XMIN        equ     24 ; Destination blit rectangle minimum X      .w
DST_YMIN        equ     26 ; Destination blit rectangle minimum Y      .w

WIDTH           equ     28 ; Width of blit rectangle                   .w
HEIGHT          equ     30 ; Height of blit rectangle                  .w
PLANES          equ     32 ; Number of planes to blit                  .w

BLiT_iT:

        lea     BLiTTER,a5          ; a5-> BLiTTER register block

*
* Calculate Xmax coordinates from Xmin coordinates and width
*
        move.w  WIDTH(a4),d6
        subq.w  #1,d6               ; d6<- width-1

        move.w  SRC_XMIN(a4),d0     ; d0<- src Xmin
        move.w  d0,d1
        add.w   d6,d1               ; d1<- src Xmax = src Xmin + width-1

        move.w  DST_XMIN(a4),d2     ; d2<- dst Xmin
        move.w  d2,d3
        add.w   d6,d3               ; d3<- dst Xmax = dst Xmin + width-1
```

```
*
* Endmasks are derived from source Xmin mod 16 and source Xmax mod 16
*
        moveq.l #$0F,d6             ; d6<- mod 16 mask

        move.w  d2,d4               ; d4<- DST_XMIN
        and.w   d6,d4               ; d4<- DST_XMIN mod 16
        add.w   d4,d4               ; d4<- offset into left end mask tbl
        move.w  lf_endmask(pc,d4.w),d4 ; d4<- left endmask

        move.w  d3,d5               ; d5<- DST_XMAX
        and.w   d6,d5               ; d5<- DST_XMAX mod 16
        add.w   d5,d5               ; d5<- offset into right end mask tbl
        move.w  rt_endmask(pc,d5.w),d5 ; d5<- inverted right end mask
        not.w   d5                  ; d5<- right end mask

*
* Skew value is (destination Xmin mod 16 - source Xmin mod 16)
* && 0x000F.  Three discriminators are used to determine the
* states of FXSR and NFSR flags:
*
*    bit 0          0: Source Xmin mod 16 =< Destination Xmin mod 16
*                   1: Source Xmin mod 16 >  Destination Xmin mod 16
*
*    bit 1          0: SrcXmax/16-SrcXmin/16 <> DstXmax/16-DstXmin/16
*                      Source span              Destination span
*                   1: SrcXmax/16-SrcXmin/16 == DstXmax/16-DstXmin/16
*
*    bit 2          0: multiple word Destination span
*                   1: single word Destination span
*
*    These flags form an offset into a skew flag table yielding
*    correct FXSR and NFSR flag states for the given source and
*    destination alignments
*

        move.w  d2,d7               ; d7<- Dst Xmin
        and.w   d6,d7               ; d7<- Dst Xmin mod16
        and.w   d0,d6               ; d6<- Src Xmin mod16
        sub.w   d6,d7               ; d7<- Dst Xmin mod16-Src Xmin mod16
*                                   ; if Sx&F > Dx&F then cy:1 else cy:0
        clr.w   d6                  ; d6<- initial skew flag table index
        addx.w  d6,d6               ; d6[bit0]<- intraword alignment flag

        lsr.w   #4,d0               ; d0<- word offset to src Xmin
        lsr.w   #4,d1               ; d1<- word offset to src Xmax
        sub.w   d0,d1               ; d1<- Src span - 1

        lsr.w   #4,d2               ; d2<- word offset to dst Xmin
        lsr.w   #4,d3               ; d3<- word offset to dst Xmax
        sub.w   d2,d3               ; d3<- Dst span - 1
        bne     set_endmasks        ; 2nd discriminator is one word dst
```

```
* When destination spans a single word, both end masks are merged
* into Endmask1.  The other end masks will be ignored by the BLiTTER

        and.w   d5,d4                   ; d4<- single word end mask
        addq.w  #4,d6                   ; d6[bit2]:1 => single word dst

set_endmasks:

        move.w  d4,Endmask1(a5)         ; left end mask
        move.w  #$FFFF,Endmask2(a5) ; center end mask
        move.w  d5,Endmask3(a5)         ; right end mask

        cmp.w   d1,d3                   ; the last discriminator is the
        bne     set_count               ; equality of src and dst spans

        addq.w  #2,d6                   ; d6[bit1]:1 => equal spans

set_count:

        move.w  d3,d4
        addq.w  #1,d4                   ; d4<- number of words in dst line
        move.w  d4,X_Count(a5)          ; set value in BLiTTER

* Calculate Source starting address:
*
*   Source Form address                 +
*  (Source Ymin * Source Form Width) +
* ((Source Xmin/16) * Source Xinc)

        move.l  SRC_FORM(a4),a0         ; a0-> start of Src form
        move.w  SRC_YMIN(a4),d4         ; d4<- offset in lines to Src Ymin
        move.w  SRC_NXLN(a4),d5         ; d5<- length of Src form line
        mulu    d5,d4                   ; d4<- byte offset to (0, Ymin)
        add.l   d4,a0                   ; a0-> (0, Ymin)

        move.w  SRC_NXWD(a4),d4         ; d4<- offset between consecutive
        move.w  d4,Src_Xinc(a5)         ;      words in Src plane

        mulu    d4,d0                   ; d0<- offset to word containing Xmin
        add.l   d0,a0                   ; a0-> 1st src word (Xmin, Ymin)

* Src_Yinc is the offset in bytes from the last word of one Source
* line to the first word of the next Source line

        mulu    d4,d1                   ; d1<- width of src line in bytes
        sub.w   d1,d5                   ; d5<- value added to pointer at end
        move.w  d5,Src_Yinc(a5)         ;      of line to reach start of next

* Calculate Destination starting address

        move.l  DST_FORM(a4),a1         ; a1-> start of dst form
        move.w  DST_YMIN(a4),d4         ; d4<- offset in lines to dst Ymin
        move.w  DST_NXLN(a4),d5         ; d5<- width of dst form
```

```
        mulu    d5,d4               ; d4<- byte offset to (0, Ymin)
        add.l   d4,a1               ; a1-> dst (0, Ymin)

        move.w  DST_NXWD(a4),d4     ; d4<- offset between consecutive
        move.w  d4,Dst_Xinc(a5)     ;      words in dst plane

        mulu    d4,d2               ; d2<- DST_NXWD * (DST_XMIN/16)
        add.l   d2,a1               ; a1-> 1st dst word (Xmin, Ymin)

* Calculate Destination Yinc

        mulu    d4,d3               ; d3<- width of dst line - DST_NXWD
        sub.w   d3,d5               ; d5<- value added to dst pointer at
        move.w  d5,Dst_Yinc(a5)     ;      end of line to reach next line

* The low nibble of the difference in Source and Destination alignment
* is the skew value.  Use the skew flag index to reference FXSR and NFSR
* states in skew flag table.

        and.b   #$0F,d7                 ; d7<- isolated skew count
        or.b    skew_flags(pc,d6.w),d7  ; d7<- necessary flags and skew
        move.b  d7,Skew(a5)             ; load Skew register

        move.b  #mHOP_Source,HOP(a5)    ; set HOP to source only
        move.b  #3,OP(a5)               ; set OP to "replace" mode

        lea     Line_Num(a5),a2         ; fast ref to Line_Num register
        move.b  #fLineBusy,d2           ; fast ref to LineBusy flag
        move.w  PLANES(a4),d7           ; d7 <- plane counter
        bra     begin
```

```
*          T h E   s E t T i N g   O f   S k E w   F l A g S
*
*
* QUALIFIERS    ACTIONS            BITBLT DIRECTION: LEFT -> RIGHT
*
* equal Sx&F>
* spans Dx&F FXSR NFSR
*
* 0     0     0    1  |..sssssssssssss|sssssssssssss..|
*                     |......ddddddddd|ddddddddddddddd|dd............|
*
* 0     1     1    0  |......ssssssssss|ssssssssssssssss|ss............|
*                     |..ddddddddddddd|ddddddddddddd..|
*
* 1     0     0    0  |..sssssssssssss|sssssssssssss..|
*                     |...ddddddddddddd|ddddddddddddd.|
*
* 1     1     1    1  |...ssssssssssss|ssssssssssssss.|
*                     |..ddddddddddddd|ddddddddddddd..|
```

```
skew_flags:

        dc.b    mSkewNFSR           ; Source span < Destination span
        dc.b    mSkewFXSR           ; Source span > Destination span
        dc.b    0                   ; Spans equal Shift Source right
        dc.b    mSkewNFSR+mSkewFXSR ; Spans equal Shift Source left

* When Destination span is but a single word ...

        dc.b    0                   ; Implies a Source span of no words
        dc.b    mSkewFXSR           ; Source span of two words
        dc.b    0                   ; Skew flags aren't set if Source and
        dc.b    0                   ; Destination spans are both one word
```

```
next_plane:

        move.l  a0,Src_Addr(a5)     ; load Source pointer to this plane
        move.l  a1,Dst_Addr(a5)     ; load Destination ptr to this plane
        move.w  HEIGHT(a4),Y_Count(a5) ; load the line count

        move.b  #mLineBusy,(a2)     ; <<< start the BLiTTER >>>

        add.w   SRC_NXPL(a4),a0     ; a0-> start of next src plane
        add.w   DST_NXPL(a4),a1     ; a1-> start of next dst plane


* The BLiTTER is usually operated with the HOG flag cleared.
* In this mode the BLiTTER and the ST's cpu share the bus equally,
* each taking 64 bus cycles while the other is halted.  This mode
* allows interrupts to be fielded by the cpu while an extensive
* BitBlt is being processed by the BLiTTER.  There is a drawback in
* that BitBlts in this shared mode may take twice as long as BitBlts
* executed in hog mode.  Ninety percent of hog mode performance is
* achieved while retaining robust interrupt handling via a method
* of prematurely restarting the BLiTTER.  When control is returned
* to the cpu by the BLiTTER, the cpu immediately resets the BUSY
* flag, restarting the BLiTTER after just 7 bus cycles rather than
* after the usual 64 cycles.  Interrupts pending will be serviced
* before the restart code regains control.  If the BUSY flag is
* reset when the Y_Count is zero, the flag will remain clear
* indicating BLiTTER completion and the BLiTTER won't be restarted.
*
* (Interrupt service routines may explicitly halt the BLiTTER
* during execution time critical sections by clearing the BUSY flag.
* The original BUSY flag state must be restored however, before
* termination of the interrupt service routine.)


restart:

        bset.b  d2,(a2)             ; Restart BLiTTER and test the BUSY
        nop                         ; flag state.  The "nop" is executed
        bne     restart             ; prior to the BLiTTER restarting.
*                                   ; Quit if the BUSY flag was clear.

begin:  dbra    d7,next_plane

        rts
```

Appendix B -- Extended ST BIOS Blit Configuration

0x40 Blitmode - Get/Set Blit Configuration

Synopsis:
```
WORD Blitmode(flag)
WORD flag;
```

Extended ST BIOS (trap #14) function number 0x40 (64 decimal) gets and sets the blit configuration. If FLAG is -1 (0xffff), then no set operation is performed, and the current blit configuration is returned. If FLAG is not -1, then the blit configuration is set as follows:

bit 0:	0: set blit mode to soft (use software) 1: set blit mode to hard (use BLiTTER)
bits 1..14:	undefined, reserved
bit 15:	must be zero

The previous blit configuration is returned in the low word of D0. The fields are:

bit 0:	0: blits are being done in software 1: blits are being done in hardware
bit 1:	0: no BLiTTER is available 1: a BLiTTER is installed in the system
bits 2..14:	undefined, reserved, may be zero or one on return.
bit 15:	always returned as zero

If an attempt is made to set the blit mode to "hard" on a system that does not contain a BLiTTER, the mode is forced to "soft".

The reserved fields are for future blit capabilities and other graphics chips. They should be treated as "don't care" fields and should be maintained (intact) because they will acquire meaning in the future.

This call works on all ROM versions of the operating system.

EXAMPLE CALL USING C

```
        #define Blitmode(a) xbios(64,a)

        curmode = Blitmode(-1);          /* get current mode */
        Blitmode(curmode | 1);           /* turn on BLiTTER */
        do_stuff();                      /* ... do some processing */
        Blitmode(curmode);               /* restore blit state */
```

EXAMPLE CALL USING 68000 ASSEMBLY

```
        move.w  #-1,-(sp)                ; D0 = Blitmode(-1)
        move.w  #$40,-(sp)               ;
        trap    #14                      ;
        addq    #4,sp                    ;
        move.w  d0,-(sp)                 ; save old blit state
        or.w    #1,d0                    ; make sure it's on
        move.w  d0,-(sp)                 ; Blitmode(D0)
        move.w  #$40,-(sp)               ;
        trap    #14                      ;
        addq    #4,sp                    ;
*
*       ... do some processing
*
        move.w  #$40,-(sp)               ; restore old blit mode
        trap    #14                      ;   from stacked old-state
        addq    #4,sp                    ;
```

Appendix C -- References

[1] Rob Pike, Leo Guibas, and Dan Ingalls, 'SIGGRAPH'84 Course Notes: Bitmap Graphics', AT&T Bell Laboratories 1984.

[2] William Newman and Robert Sproull, 'Principles of Interactive Computer Graphics', McGraw-Hill 1979, Chapter 18.

[3] John Atwood, '16160 RasterOp Chip Data Sheet', Silicon Compilers 1984. See also 'VL16160 RasterOp Graphics/Boolean Operation ALU', VLSI Technology 1986.

[4] Adele Goldberg and David Robson, 'Smalltalk-80: The Language and its Implementation', Addison-Wesley 1983, Chapter 18.