Programmer's Guide to Orbit, the ROS Adapter, and the Dover Printer

by **Bob Sproull**
revised by Dan Swinehart

April 14, 1978

This report presents the Orbit, ROS adapter and Dover printer hardware from the programmer's point of view. It is intended to contain "all you need to know" to write printing programs using the gear. In addition to information about control of the hardware, it gives various techniques and tricks that were either planned or discovered.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Introduction

The hardware described in this report is intended for printing pages consisting primarily of text and lines on raster-scanned printing devices. There are two main tasks that this collection of hardware performs: (1) *image generation*, in which a binary representation of a video signal is prepared at high speed, and (2) *transmission* of that video signal to a raster output scanner (ROS).

Although the hardware can be adapted for other uses, the report takes the view that the following configuration is being used:

> Alto II. We begin with an Alto II computer, with which the reader is assumed to be familiar (see the "Alto Hardware Manual"). We add:
>
> Orbit, a four-card hardware device that plugs into the Alto II. The Alto II requires a specially-wired backpanel to accommodate the Orbit cards (some Alto II's are manufactured with this backpanel already arranged). Orbit provides facilities for image generation: the resulting raster image can be returned to the Alto memory or can be forwarded via a "9-wire standard ROS interface" to:
>
> TTL ROS adapter, a two-board hardware module responsible for orderly communication among the Orbit, ROS optics head and printing engine. This is usually mounted in, and connected to:
>
> The Dover printer, a Xerox 7000 copier with a specially-designed laser-scanned imaging head on it. Properly controlled, this engine delivers 1 page/second (paper speed is 10 inches per second). A modification is possible (the "extended Dover printer") that slows the transport speed to 5 inches/second and page delivery to 1 page/2 seconds.

There are numerous variations on this basic arrangement. Because the Alto II/Orbit pair communicates with the adapter using a standard interface, it is capable of operating with other adapters and printing engines that conform to the standard. Also, the TTL ROS adapter can be used to control other printing engines. For example, a common alternative is:

> The Sequoia printer, a Xerox 3100 copier with a ROS head on it. It delivers 1 page/3 seconds (paper speed is 3.46 inches/second).

The basic idea of the entire arrangement is that a program in the Alto II can operate all these devices to cause pages to emerge from a printer. The remainder of this document will by and large assume that we are operating the standard Dover printer (10 inches/second). The basic ideas will remain the same when operating other printers, but many of the timing details will change.

For the purposes of discussion, we will assume that the printer scans the page in *landscape mode* (mode=8, in the Press terminology). Scan-lines run from bottom to top of the page (when the page is held in the normal portrait orientation); the first scan-line is at the left-hand side of the page. Orbit does not "know" that the video signal it is generating will be used to scan in landscape or in portrait mode, but we need to stick to one convention for the purposes of exposition.

The Orbit, ROS adapter and Dover printer system is quite tightly integrated; consequently, explication of the system is somewhat difficult. The reader is urged to bear with us, to skip sections that become obscure, and to read the document several times before abandoning all hope! For a first reading, the following sections are recommended:

> Section 1, through 1.4

Section 2, through 2.4
Section 3

People

Numerous individuals and groups have contributed to the Dover program: the design and fabrication of Orbit, the ROS adapter and the Dover printer and ROS. This document is only a condensation of their designs and ideas. The overall Dover Program was managed by John Ellenby. The Dover Printer was designed by Gary Starkweather, Gary Swager, Tibor Fisli, Ed Wakida and Bobby Nishimura with contributions on producibility from Abbey Silverstone and on mechanical and industrial design from members of EOD/SPG and others on the program team. The ROS adapter was designed by Ron Rider, Ken Pier and Ron Freeman. Orbit grew out of a suggestion by Butler Lampson, and was designed by Severo Ornstein, Jim Leung and Bob Sproull. The standard microcode and test software were devised by Bob Sproull. In the closing months of the program, Gary Swager played a key role in coordinating the system integration of the Dover printer electronics, electro-optics, and the 7000 base. Dover ROS equipment is manufactured under the direction of Tibor Fisli and the Dover System is manufactured and integrated by SPG under the direction of Doug Stewart who also supervised the design and engineering contributions of SPG. Bob Sproull and Dan Swinehart produced the Spruce software that provides printing service using Orbit devices.

Numerous organizations cooperated very effectively on the Dover project. The people mentioned above came from PARC/CSL, PARC/OSL, PARC/SSL, SDD/SD, and EOD/SPG. The project was feasible in part because flexible funding was provided by the PARC Office of the Center Director.

1. Orbit

Orbit's main chore is that of *image generation*: it is designed to *merge rasters* at very high speed. Normally, Orbit merges raster definitions of individual character shapes into a very large raster that describes an entire page of text.

By a *raster* we mean a rectangular region of a raster-scanned image, in this case an array of 1-bit values. A raster is characterized by its *width* and *height* (see Figure 1-1). Although there are many ways that the data bits within a raster might be represented, we shall be interested in only the simplest: we record a *stream of bits*, generated by scanning the raster in a prescribed (conventional) manner. We shall assume that rasters are scanned from bottom to top, and then from left to right. Figure 1-1 shows a tiny raster, the bit stream that results from scanning it, and the blocking of that bit stream into 16-bit words suitable for recording in Alto memory.

If we were to generate a raster representation of an entire page, and save it in memory, we would be overwhelmed by its size. A page 8.5 by 11 inches scanned at 350 bits/inch requires $11.5 * 10^6$ bits, or some 710,000 words of Alto memory! Therefore, we are forced to generate the page raster *incrementally*, using smaller buffers. If we wish to avoid a full-page buffer, we must of course generate the scan-lines of the raster in the same order as the ROS device accepts them.

The unit of buffering in Orbit is the *band*, a group of 16 consecutive scan-lines. We will want to generate band images in the same order that the ROS accepts the scan-lines that comprise the bands. In the case of Dover, which scans in landscape mode, we will generate first the band corresponding to the left-most 16 scan-lines on the page, then the next band to the right, etc. Orbit's task is to prepare the raster describing each band faster than the ROS yanks band rasters away in order to print them!

Orbit contains *two* buffers, each of which holds one band; the buffers are labeled A and B. While one buffer is being used to prepare an image (the *image* buffer), the other, which contains an already-prepared image, is being sent to the ROS (the *output* buffer). After the ROS is finished accepting the band raster, the roles of the two buffers are switched. Each buffer contains 16 scan-lines; each scan-line is 4096 bits long (see Figure 1-2). We can thus think of a band buffer as a two-dimensional array of bits, in which the *x* dimension (0 through 15) indexes scan-lines, and the *y* dimension (0 through 4095) indexes bits within a scan-line. If the total height of the raster on the page is less than 4096 bits, the *upper region of the band should be used* (that is, the region with larger *y* addresses). The reason for this will appear below.

1.1 Image generation

Orbit excels at merging rasters represented as bit streams in the Alto memory into the *image* band buffer. Figure 1-3 illustrates one case: a character is represented by a raster in memory, characterized by its *height* and *width*. It is to be merged into the band by placing its lower left corner at location (*x*,*y*) in the band. This is effected as follows:

1. Alto microcode passes to the Orbit hardware the four parameters *x, y, height* and *width.*

2. Alto microcode starts passing to Orbit the successive 16-bit words representing the bit stream for the source raster. Orbit takes each word, positions it on the proper scan-line, and at the proper vertical position, and merges the 16 bits into the raster already saved in the band. Each time Orbit finishes work on a scan-line, it increments the *x* position to move to a new scan-line, and decrements a *width* counter that counts remaining scan-lines in the source raster.

3a. Eventually, (at least in this example) the width counter reaches zero, indicating that the source raster has been exhausted. The Alto microcode loop that is repeatedly passing raster bits to Orbit is terminated when the width reaches zero.

It commonly happens that the width of a character exceeds the width of a band, and thus the character extends into subsequent bands. In this case, we need another way to terminate the loop:

3b. Eventually, the last scan-line in the band (*x*=15) has been processed, even though the (character) width counter has not been decremented to zero. We must terminate that loop that passes raster bits to Orbit, but we must remember enough information so that we can resume processing of the same character when it comes time to work on the next band.

The "left over" information that must be remembered for the next band consists of two sorts of information: basic information about the character (the *height* of the source raster and the *y* coordinate that determines where the bottom of the source raster is to be placed in the band) and a description of the portion of the character remaining to be generated (a new value of the *width* counter, representing the unprocessed width remaining, and a new pointer into the bit stream for the source raster, which describes where to resume taking bits). The leftover source raster pointer requires 20 bits, and consists of two parts: the most significant 16 bits specify which Alto memory word is the first to take up next time, and the low order 4 bits specify which bit within the word should be the first to be processed.

So Orbit gives us help with the inner-most loop: positioning the character raster on the band raster, and merging the bits into it. Alto microcode, however, is responsible for an orderly processing of characters, for fetching source raster words and passing them to Orbit, for saving "left over" information, and for interpreting this information properly when the next band begins.

The details of the "merge" operation performed by Orbit offer some flexibility. A band buffer starts out with all bits set to 0--this is actually accomplished by clearing the words of the band as they are read out and passed either back to the Alto or on to the ROS. The "inner loop" of the merge operation is responsible for changing a particular bit in the band buffer (BAND[x,y]), based on the value of a particular bit of the source raster bit stream (SOURCEBIT). The algorithm is:

if SOURCEBIT=1 *then* BAND[*x,y*] _ INK[*x, y* mod 16]

where INK is a 16 by 16 bit memory in the Orbit hardware that can be set by the Alto. The intent of the ink memory is to permit "gray" colors to be created by spatial halftones. In most cases, characters will want to be "black," and the INK memory should contain all 1's. In light of the above algorithm, the proper way to view the input raster is: *it is a mask, with 1 bits everywhere the "character" form lies.* It is *not* correct to think of the 1 bits in the source raster as "data" bits--the INK memory provides the data bits.

There are some special cases of the Orbit image-generation technique that deserve mention. First, it is possible to have a source raster with *width*=15 and *height*=4096. This is a useful mode of operation when a binary image has been pre-computed, and Orbit is being used simply as a pipe to shovel the image toward the ROS. Second, there is at least one case when it is senseless to have an actual physical representation for the source raster. This occurs when merging "rules" (or "lines") into the band raster: in this case, the raster representation of the input bit stream consists of an endless string of 1's, and we do not need to actually store this string in memory. For rules, then, it is more convenient to provide the four parameters (*x, y, width, height*), but to omit the memory for the bit stream, and simply ship -1's (words with all 16 bits equal to 1) to Orbit.

1.2 Reading a buffer

While the *image* band buffer is being prepared with an image, the *output* band is being read out, either to the ROS adapter or back into Alto memory. The read-out mechanisms are similar, and require some explication. The basic read-out operation cycles through all 16 scan-lines (*x*=0 to 15), and delivers information from each scan-line beginning with *y*=FA*16, and ending with *y*=4095 (FA, which denotes "first address," is a number in the range 0 to 255 that can be set by the Alto). After the last bit (4095) of the last scan-line (*x*=15) has been read out, Orbit will "switch the buffers" (i.e., interchange the role of buffer A and buffer B). The next bit to be read out will come from the other buffer.

The FA mechanism has a minor effect on the way images should be generated. If the receiver of the video (usually the printer) is instructed to place the first bit of each scan-line at the bottom of the page, the next slightly above it, and so forth, then the relation between vertical position on the page (*yp*) and address of the bit in Orbit's buffer (*y*) is: $y = yp + (FA*16)$. Consequently, the number (FA*16) must be added to each desired page position (*yp*, measured in bits) to determine the proper *y* address to provide to Orbit. (Note: This addition can be easily excluded from the inner loop by making modifications to *y* incrementally, spacing appropriately for each character encountered.)

The act of reading a buffer also writes zeroes into it: this is an economical mechanism for clearing a buffer. The normal practice, therefore, is to read a buffer thoroughly simply in order to clear it. After the buffer is read, a "buffer switch" happens, and the cleared buffer becomes the *image* band buffer.

The most common use of Orbit is to drive a ROS printer. In this case, FA is simply set so that the upper portion of the buffer which is used corresponds to the desired image height for the entire page. In order to clear both Orbit buffers initially, we arrange two dummy bands of image information that will lie *off* the actual page image (i.e., to the left of the page, on Dover). Thus the first two buffers read, which may contain garbage left behind from previous operations and from lack of refreshing, will not be visible on the page. When used to drive a ROS (which requires FA to be a multiple of 4 in order to communicate video data sucessfully to the ROS adapter), Orbit can deliver video data at a rate of 23 MBits/sec.

The other common use of Orbit is to take advantage of its fast image-generating capability, but to return a completed band to the Alto memory. Details of this operation are given in section 1.8.

1.3 Communicating with the ROS adapter

Orbit also provides functions for controlling the ROS adapter and its associated printer. Two completely independent mechanisms are used for sending control information and for reading ROS (and adapter) status. Orbit will accept 16-bit commands from the Alto and forward them to the adapter. It is also constantly reading the status line returning from the adapter, and is capable of reporting 256 bits of status generated by the adapter.

1.4 Orbit Programming -- the common cases

The next several sections of this guide describe the operation of Orbit undertaken with the "standard microcode," and then the regimen for writing microcode if you so desire. This section describes those elements of Orbit that must be understood to operate the hardware using either the standard microcode or specially-written code.

The control of Orbit is accomplished with one Alto microtask (task 1, next to lowest priority), and an associated collection of microcode that executes within that task. Remember that the Alto must be "bootstrapped" after loading the microcode into the RAM in order to initiate task 1's program counter to point to the proper RAM routine. (For more information about

microcoding and bootstrapping, see the Alto Hardware Manual.)

The record of activity of the Orbit task is quite complicated. The task is initially inactive: the Alto program must initiate Orbit activity by issuing StartIO(4), which signals the Orbit hardware to activate its task. During the active period, the details of the microcode functions sent from the Alto to Orbit will govern exactly when the Orbit task is run. However, at the termination of a sequence of commands the microcode usually instructs Orbit to return to the initial, inactive, state. (This last is simply a convention; the "standard microcode" normally uses this convention, but has an exception as well.)

Standard microcode

The remainder of this section describes simple use of 6 functions of the standard Orbit microcode. (The standard microcode includes many additional functions for debugging and for more exotic uses of Orbit. These are taken up in subsequent sections.) The standard microcode was written to accomodate most Orbit chores, including normal printing tasks and hardware diagnosis. The microcode source files can be found on [IVY]<SPRUCE>SpruceMc.Dm.

When the Orbit microcode is activated, it figures out what to do by reading a *control table*. To execute a command, a pointer to the control table (which must begin at an even address) is placed in location 720b, and StartIO(4) is executed. When execution of the command is complete, the Orbit task stores 0 in location 720b, an event that can be detected by the Alto program. Note that during execution of an Orbit command, the Alto emulator program continues to run--the Orbit task and the main program are asynchronous. To allow Orbit maximum access to the Alto microprocessor, it is advisable to curtail activities of other microtasks while printing is in progress. The display should certainly be turned off; stopping disk or Ethernet activity is less beneficial.

The control table is 11 words long, and has conventional names for each word:

Directive
Argument1
Argument2
nBandsMinus1
FAword
LoTable
FontTable
NewCharPointer
Result
OrbitStatusResult
CurrentCopy

The *Directive* is a small integer that indexes the various operations that can be performed by the microcode; each operation is given a name of the form **Program**xx. When a command terminates, the control table entry OrbitStatusResult is loaded with the inclusive "or" of Orbit hardware status (**OrbitStatus**, bits 8-15) and microcode status (FirmwareStatus, bits 0-4), described below. If an operation returns a result, it is usually stored in the Result entry of the control table.

ProgramControl (directive=0). Argument1 is passed to Orbit as a control word. In the simple situation we are considering here, this function is used only to reset the Orbit hardware entirely (Argument1=1).

ProgramXY (directive=3) and **ProgramInk** (directive=7). In the simple situation we are considering, it is necessary to set the INK memory to contain all 1's so that all characters will appear black on a white background. This can be achieved during initialization with a simple loop that uses **ProgramXY** to set *x*, and **ProgramInk** to set the 16 corresponding bits of the INK memory:

```
for i=0 to 15 do begin
    Execute ProgramXY with Argument1=i * 4096 (i.e., i in leftmost 4 bits)
    Execute ProgramInk with Argument1=-1
end
```

ProgramROSCommand (directive=9). This operation sends Argument1 (16 bits) to Orbit to be forwarded to the adapter as a command. Adapter commands are discussed further in section 2.

ProgramROSStatus (directive=14). This operation reads an entire word (16 bits) of ROS status. If we view the 256 status bits as lying in 16 16-bit words, numbered 0 to 15, then this function stores in Result the current value of status word *n*, where Argument1 = *n* * 1024.

ProgramGeneratePage (directive=12). This function is the only complicated one, and is responsible for image-generation and printing of an entire page. The details are presented below.

The page-generation function

The **ProgramGeneratePage** function is the main "workhorse," and contains the necessary facilities to keep a printer running at high speed. It depends on a data structure for describing the placement of characters on the page and for encoding the rasters that are to be printed for each character (the font). Building this data structure will require a pass over the each page before it is printed (more on this later). Figure 1-4 summarizes these data structures, which are explained in the following paragraphs.

Fonts. The microcode believes that there is only one font, but it may contain an enormous number of characters (up to 2^{15} if you can figure out where to find memory for them!). Each character in the font is represented by a contiguous block of memory words, formated as follows:

Word 0:	-Height (in bits) of the character. Heights of 1 to 4095 are legal.
Word 1:	Width-1 (in scan-lines). Widths of 1 to 4096 are legal.
Word 2-*n*:	Bit stream for the encoding of the character raster

Word 0 must lie on an even memory address. The sample character given in Figure 1-1 would therefore be represented as:

Word 0:	-4
Word 1:	4
Word 2:	103126b
Word 3:	100000b

To verify that you understand this encoding, you should be able to compute *n* (answer: `i`(Height*Width)/16`j`+1).

The font is indexed with a table: if the pointer to the base of the table is denoted by FontTableX, then the contents of memory location FontTableX+*cc* points to Word 0 of the character representation for character with code *cc*.

Band Lists. The most interesting piece of data structure is the *band list*, an encoding of where instances of various characters should appear on the printed page. This list is divided into segments, one for each band. Each segment gives information about the characters that *begin* in the corresponding band; Orbit microcode will handle the details associated with continuing the characters into subsequent bands if necessary.

A band list segment is a table of entries of one of four kinds:

1. Character. A character to be placed on the page is described by a two-word entry:

Word 0[0]	This bit must be 1.
Word 0[1-15]	Character code. This number is used to index the font table to find a pointer to the character encoding for the character to be printed.
Word 1[0-3]	XOffset. This gives the scan-line, within the band, at which the left-most edge of the character raster should appear.
Word 1[4-15]	Y. This entry gives the *y* address at which the bottom edge of the character raster should appear.

2. Rule. A rule to be placed on the page is described by a four-word entry:

Word 0	This word must equal 1.
Word 1[0-3]	XOffset. This gives the scan-line, within the band, at which the left edge of the rule should appear.
Word 1[4-15]	Y. This entry gives the *y* address of the bottom edge of the rule.
Word 2	-Height. This entry gives the negative of the number of bits high the rule should be. Heights of 1 to 4095 are legal.
Word 3	Width-1. This gives the width, in scan-lines, of the rule. Widths of 1 to 4096 are legal.

3. Jump. This entry provides a conditional method for obeying or ignoring selected band list entries, to implement the "only on copy" feature in Press format files (see further discussions below.)

Word 0	This word must equal 4+(Copy*32), where Copy is the copy number in which the following entries are to be printed. A jump will occur, omitting these entries, if Word 0 is *not equal to* the CurrentCopy field in the control table. A version of this command must appear in every band in which information conditioned on this copy number occurs. Copy must be a positive integer less than 1024.
Word 1	Jump distance. This must be equal to the size, in words, of the entries to be omitted when the inequality holds.

4. End-of-Band. After the last entry for the band, a two-word terminator is placed. Both words should be 0.

A complete band list, consisting of a contiguous sequence of band list segments (one for each band), describes an entire page. In the discussion below, we shall assume that the band list for a page is pointed to by the pointer BandListPointer; the band list must begin at an even address.

In most cases, constructing the band list from a document description will require an initial pass over the document description. One of the functions of this pass will be to map font descriptions and character codes into the one-dimensional font representation used in the structures above (i.e., a single 15-bit character code). Another function of this pass is to perform a *sort*: it will be necessary to process characters as they are extracted from a document description, determine the *x* and *y* positions of the lower left corner of the character raster by consulting a font description, and build a 2-word character entry. These entries will then need to be sorted by band number (a bucket sort is particularly reasonable for this task) in order to re-assemble the character entries into band list segments.

Left Over Table. The Orbit microcode will normally need access to a region of memory that can be used to save left-over information needed to resume image generation of characters that extend beyond one band. The discussion below assumes that the pointer LoTablePointer

points to such a table; the table must start on an even address. The left-over table can be reset to the empty state by setting the first word to 0. Each left-over character in the table requires 4 words. Further understanding of the format of the left-over table is not essential to operate **ProgramGeneratePage**, but may be helpful for debugging (see Section 1.10).

Arguments. This function performs image-generation chores for a number of bands, and in addition forwards the image to the ROS adapter for printing by turning on SLOTTAKE. The arguments in the Orbit control table are:

Argument1 = Pointer to adapter command table (must be even; see below)
Argument2 = Timeout count, in units of 2 ms.
nBandsMinus1 = number of bands to generate, minus 1
FAword = FA*256 (i.e., FA must be in the left byte)
LoTable = LoTablePointer (left-over pointer, must be even)
FontTable = FontTableX+100000b (pointer to font table, offset by 100000b)
NewCharPointer = BandListPointer-1 (BandListPointer must be even)
CurrentCopy = 4 + (current copy number)*32

The function will attempt to generate images for the specified number of bands; of course the band list pointed to by NewCharPointer+1 must contain an adequate number of band list segments. The value in the nBandsMinus1 entry will be decremented at the end of each band, allowing the emulator to track Orbit's progress. The timeout count is used whenever Orbit is waiting for the ROS adapter to finish reading the *output* buffer: if the timeout is exceeded, the microcode completes early (as if nBandsMinus1 were so small that it did not include the next band) and sets the firmware status bit TIMEOUTmc (bit 1) in the OrbitStatus word of the control table. The most common reason for timeout is the failure of page sync to arrive: these issues cannot be fully discussed until the operation of the adapter is described in some detail in section 3.

After the last band is generated, the **ProgramGeneratePage** function completes (i.e., sets location 720b to zero), even though the ROS adapter has probably not finished taking image data. In fact, the adapter can continue to request and receive data *ad infinitum*, but the data will eventually become all zero (recall that reading an output buffer zeroes it, and since the microcode is no longer invoking image-generation functions, no new image data are being added to the Orbit buffers). This is a convenient way to finish out the printing of a page with white borders. To halt the taking of image data, Orbit can be reset. After Orbit is reset, the ROS adapter will simply repeat the most recent video data it received correctly.

The adapter command table (pointed to by Argument1) is used to encode commands that should be sent to the ROS adapter *during* the generation of a page. Such a mechanism is necessary because the Alto program would not otherwise be able to signal the printer device during the page generation (because the **OrbitROSCommand** can be issued only by the Alto task assigned to Orbit, which is occupied with page generation). The table must begin on an even address, and contains two-word entries in the following format:

Word 0[0-3]	AdapterCommandCode
Word 0[4-12]	whenBandMinus1. This entry gives the band number before which the corresponding command will be executed. The first band is numbered nBandMinus1, then nBandMinus1-1, etc. down to 0.
Word 1	AdapterArgument

The table will be processed in the order given, i.e., the whenBandMinus1 entries should be arranged in decreasing order. An appropriate termination entry is Word 0=177777b, which specifies an unreasonably large value for whenBandMinus1. There are presently three AdapterCommandCodes defined:

0. Send AdapterArgument to the adapter as a command (i.e., **OrbitROSCommand _** AdapterArgument).

1. Read 4 bits of adapter status; the four bits are numbered 4*n* to 4*n*+3, where AdapterArgument=*n**256 (*n* is in the left byte). The resulting status is read and stored in bits 12-15 of the memory word that previously held AdapterArgument. (The exact operations are: **OrbitControl** _ AdapterArgument; AdapterArgument _ **OrbitStatus**.)

3. Wait for AdapterArgument/3 microseconds in a tight loop. This command is provided in case it is absolutely essential that two adapter commands be issued before the same band. In this case, we must wait 60 microseconds between issuing adapter commands to assure proper communications with adapter.

If Jump codes appear in the band list, the CurrentCopy entry must be maintained. This word must contain an encoding, as shown, of the number of times, plus one, that the current document has been printed during the current run; that is, the number of the current copy.

When page generation has terminated, the Result word in the control table will contain the value that nBandsMinus1 had when word 8, bit 3 of adapter status was last on during image generation; this word will contain the original nBandsMinus1, plus one, if the bit never comes on. This very special feature assists with detecting an important but elusive timing signal in the Dover printer (see the description of *Count-H* in the Dover section.)

Managing the data structures

During the generation of a page, the three data structures (*font, band list* and *left over table*) must of course be resident in memory. In order to keep a ROS running at high speed, the font and band list structures for each page must be present in memory at the moment it is necessary to begin generating the image for the corresponding page. To do this requires a buffering strategy that is willing to read information for page *i* from a disk while page *i*-1 is being imaged. (The left over table is reasonably small, and can of course be re-used for subsequent pages. Consequently, we shall omit further mention of it.)

There are basically three methods of buffering (see Figure 1-5):

Method 1. Font and band list structures for each page are generated on the disk, and read into one of two memory buffers. While page *i*-1 is being imaged, the structures for page *i* are being read. (This scheme is not a particularly attractive alternative, because the complexity of a page is severely restricted by the amount of available memory.)

Method 2. One font data structure will suffice for all the pages to be printed, and can therefore remain resident in memory. The band lists are put in two buffers: while page *i*-1 is being imaged, the band lists for page *i* are being read. This scheme is the recommended scheme whenever the page complexities allow the font and two band list buffers to fit in memory. The advantage of this scheme is that it permits printers to run at high speed, without pauses between pages.

Method 3. All of memory is used for one font structure and one band list. After page *i*-1 is imaged, it is necessary to read the font and band structures for page *i* before imaging may start for that page. If used with high-speed printers, it may be necessary to stop the printer while page *i* is read in, because otherwise there would be insufficient time to complete disk activity before imaging activity would need to begin. This scheme does have the advantage that maximally complex pages can be printed, by allocating all of memory to the structures necessary for that page.

Note that hybrid schemes may be used as well: methods 2 and 3 may be mixed, although it may be necessary to stop the printer as the method is being switched (for the same reason it may be necessary to stop the printer between pages using method 3). Planning the buffering and font makeup is one of the less pleasant tasks of the initial pass over the document that builds band lists!

The remainder of section 1 presents details that are not essential to most printing applications. Read on only if you are (1) hardy, (2) desirous of writing microcode, (3) desirous of writing Orbit test programs, or (4) need to know how to use Orbit for non-real-time image generation help.

1.5 Orbit Hardware Functions

There are various kinds of transfers of information between the Alto microprocessor and Orbit; each can transfer up to 16 bits of information. Although the transfers are actually accomplished with microcode functions interpreted by Orbit, the functions themselves are used by higher-level language programmers as well. The list below gives each transfer a name (of the form **Orbit**xx); the description assumes a 16-bit *value* is transferred (refer to Figure 1-6 for a tabular summary of this information):

Functions for status and control

OrbitControl. This function transfers 16 bits of control information to Orbit. The fields of this value are as follows:

Field	Bits	Function
auxControl	0-7	This 8-bit field has two different interpretations, depending on the setting of the WHICH field (q.v.).
ESS	8	This bit must be 1 to enable changing the SLOTTAKE setting (q.v.).
SLOTTAKE	9	This bit setting, enabled by ESS above, controls the output buffer logic. Normally (SLOTTAKE=0), Orbit will not honor video data requests coming from the adapter. As soon as SLOTTAKE is set to 1, however, output data will be passed to the adapter when it demands it.
clrBEHIND	11	This bit clears the BEHIND indicator (see below).
setGOAWAY	12	This bit controls microcode wakeups, and should normally be 0. (See section on microcoding for more information.)
WHICH	13	This bit controls the use to which the 8-bit auxControl field is put. If WHICH=0, auxControl is interpreted as an address (range 0 to 63) into the adapter status memory: when **OrbitStatus** is next interrogated, 4 status bits (with numbers 4*auxControl to 4*auxControl+3) will be reported. If WHICH=1, auxControl is used to set FA.
CLRFRESH	14	This bit controls refresh logic, and should normally be 0. (See section on microcoding for more information.)
RESET	15	This bit, if 1, will reset Orbit entirely. A reset performs at least the following functions: FA_0, SLOTTAKE_0, band buffer A is assigned to the *image* buffer, the status and control dialogues with the adapter are reset.

OrbitStatus. This function reads 16 bits of status information from Orbit into the Alto. The starred status bits below (*) are generated by firmware only in some situations. The notation a_**OrbitStatus** denotes the hardware status-reading function, and does not include the firmware status bits. The notation a_**OrbitStatus**+FirmwareStatus denotes the full status report. The status fields are:

Field	Bits	Function
IACSmc	0*	This bit is 1 if Orbit is "in a character segment," i.e., if Orbit needs more **OrbitFontData** transfers in order to complete image-generation for the current character. (This bit is not actually reported by Orbit to the Alto in this way, but is inserted by the "standard microcode.")
TIMEOUTmc	1*	(This bit is not actually reported by Orbit to the Alto, but is inserted by the "standard microcode." See the description of **ProgramGeneratePage**, above).

unstableROSmc	2*	(This bit is not actually reported by Orbit to the Alto, but is inserted by the "standard microcode." See the description of **ProgramROSStatus**, above).
earlyPageAbortmc	3*	(This bit is not actually reported by Orbit to the Alto, but is inserted by the "standard microcode.") If, at the end of a band, the adapter has stopped accepting video information, and if the BEHIND bit (below) is set, **ProgramGeneratePage** terminates early (as if it had timed out), with this bit set.
badBandEntrymc	4*	(This bit is not actually reported by Orbit to the Alto, but is inserted by the "standard microcode.") The **ProgramGeneratePage** microcode has encountered a non-character band entry that does not specify a valid operation. It has stored additional diagnostic information in fixed memory locations. This condition should only arise in instances of severe hardware or software faults. Consult the Orbit microcode listings for further information.
badROS	8	This bit is true if an error of some sort has been detected on the adapter status-reporting line. The adapter status cannot be trusted. The condition will be reset after the first properly-formatted status burst is received from the adapter.
INCON	9	This bit announces the current band buffer assignments. If it is 0, buffer A is the *image* buffer, and buffer B is the *output* buffer. Watching this bit for changes is the proper way to detect "buffer switches."
stableROS	10	This bit is true if a status burst is *not* currently arriving from the adapter. If you insist on reading some adapter status that must be from the same burst, you can check this bit before and after a read.
BEHIND	11	This bit is set when Orbit "gets behind." This will happen if the buffers switch (due to output requests) before the Alto microcode has finished sending the next band of image-generation information to Orbit (finishing is signaled to Orbit by setting GOAWAY).
ROSStatus	12-15	This field gives 4 bits of status reported by the adapter. See the description of WHICH, above.

Functions for controlling image-generation

There are a number of functions for controlling image generation. There is a reasonably standard sequence of operations for transferring source rasters into Orbit. The first three functions (**OrbitHeight**, **OrbitXY**, and **OrbitDBCWidthSet**) are used to pass to Orbit the four parameters that describe a source raster and its position in the band. The last of these (**OrbitDBCWidthSet**) sets a flag IACS ("in a character segment") that indicates Orbit is expecting source raster data. Then the **OrbitFontData** function is used to send words of source raster information to Orbit. After the character is finished (or we run into the edge of the band at *x*=15), the two functions **OrbitDBCWidthRead** and **OrbitDeltaWC** are used to extract from the Orbit hardware the necessary parameters to compose a "left over" entry so that image generation for this character can be resumed in the next band.

OrbitHeight. This command sends to Orbit a 12-bit field (*value*[4-15]) which is interpreted as the *two's complement* of the height of the source raster, in bits.

OrbitXY. This command sets *x_value*[0-3] and *y_value*[4-15]. It is therefore used to set the starting scan-line within the band (*x*) and the vertical position of the bottom of the copy of the source raster (*y*).

OrbitDBCWidthSet. This function has three purposes: First, *value*[0-3] is used to tell Orbit which bit (0 to 15) of the first word of raster data is the first bit to be examined

(i.e., it will play the role of SOURCEBIT for the setting of BAND[*x*,*y*], where *x* and *y* were set with **OrbitXY**). Second, *value*[4-15] is interpreted as the width of the source raster, *minus 1*. Third, executing this function initializes a mess of logic relating to transferring a source raster to Orbit and sets IACS ("in a character segment"). After the function is executed, it is wise to issue only **OrbitFontData** functions until the image-generation for this character terminates (i.e, IACS becomes 0). Warning: IACS is cleared by StartIO(4); consequently it is not possible to "single step" Orbit's character generation.

OrbitFontData. This function is used to send 16-bit raster data words to Orbit, usually in a very tight loop. The loop is exited when IACS becomes 0, that is, when the Orbit hardware counts the width counter down to zero, or when the *x* counter overflows (i.e., when it would count up from 15 to 16).

OrbitDBCWidthRead. This function returns from Orbit to the Alto a 16-bit value that corresponds to the value set with **OrbitDBCWidthSet**, but *after* the image-generation for this character (in this band) has completed. Thus, *value*[0-3] tells which bit should be the first to be examined when image-generation resumes in the next band. Also, *value*[4-15] contains the remaining width (*minus 1*) of the character. If *value*[4-15]=7777b, there is no remaining width, because the width counter counted down to zero (i.e., the source raster was exhausted).

OrbitDeltaWC. This function returns from Orbit to the Alto the count of the number of full 16-bit words of **OrbitFontData** that were used during the image-generation for this character (the counter is initialized to 0 when **OrbitDBCWidthSet** is issued). This allows the Alto program to compute the memory address in the source raster where generation should resume in the next band. (Note: This function is needed because it is too time-consuming to derive the count by calculation. If you really understand Orbit, you will be able to verify that DeltaWC=(Height*SL+DBC)/16, where DBC is the 4-bit value passed to Orbit with **OrbitDBCWidthSet**, and SL is [if x+Width < 16 then Width else 16-x].)

Functions for communicating with the adapter

Reading adapter status is accomplished with the **OrbitControl** and **OrbitStatus** functions (first set the status memory address, then read 4 status bits). Commands are sent to the adapter with:

OrbitROSCommand. This function sends a 16-bit command to the ROS. Another ROS command should not be issued until 60 microseconds have elapsed, to allow time for proper transmission of the command to the ROS.

Miscellaneous functions

OrbitOutputData. This function reads 16 bits of output data from Orbit into the Alto. The sequencing of words of data is described below.

OrbitInk. This function sets 16 bits of INK memory: INK[*x*,0] through INK[*x*,15], where *x* has been previously set with **OrbitXY**.

1.6 Standard microcode -- complete description

The standard microcode was introduced in section 1.4; the description of bootstrapping and of the control table apply here as well. Here we give a complete list of the functions it implements:

ProgramControl (directive=0). Argument1 is passed to Orbit as a control word: **OrbitControl _** Argument1.

ProgramOutputData (directive=1). This operation simply reads 16 bits of Orbit output data: Result _ **OrbitOutputData**.

ProgramHeight (directive=2). This operation sets the Orbit height parameter: **OrbitHeight** _ Argument1. It also returns a result: Result _ (if *image* buffer needs refreshing then -1 else 0).

ProgramXY (directive=3). This operation sets the Orbit X and Y registers: **OrbitXY** _ Argument1.

ProgramDBCWidthSet (directive=4). This operation sets the DBC and Width counters: **OrbitDBCWidthSet** _ Argument1.

ProgramFontData (directive=5). This operation transfers one word of font data to Orbit: **OrbitFontData** _ Argument1.

ProgramDeltaWC (directive=6). This operation reads the word counter: Result _ **OrbitDeltaWC**. Because each function implemented by the standard microcode reports Orbit status on completion, the **ProgramDeltaWC** function is a fine way to read Orbit status without changing any Orbit state.

ProgramInk (directive=7). This operation transfers a word to the ink memory: **OrbitInk** _ Argument1.

ProgramDBCWidthRead (directive=8). This operation reads the current settings of the DBC and Width counters: Result _ **OrbitDBCWidthRead**.

ProgramROSCommand (directive=9). This operation sends 16 bits to Orbit to be forwarded to the adapter as a command: **OrbitROSCommand** _ Argument1.

ProgramReadBlock (directive=10). This operation is used to read a sequence of **OrbitOutputData** words. Argument1 gives the number of words to read; Argument2 is the address where the first word should be stored. The function is equivalent to the following code:

```
for i=0 to Argument1-1 do
    Argument2!i _ OrbitOutputData
```

ProgramCharacter (directive=11). This operation is used to pass a source raster to Orbit at high speed, and can be described as a collection of primitive commands (note that control table entries are used in a way that makes their names meaningless):

```
OrbitHeight _ Argument1
OrbitXY _ Argument2
OrbitDBCWidthSet _ nBandsMinus1
for i=0 to Abs(LoTable)-1 do
    OrbitFontData _ FAword!i
Result _ OrbitDBCWidthRead
NewCharPointer _ OrbitDeltaWC
```

Note that this operation returns two values, one of which is stored in the NewCharPointer word of the control table (a non-standard place). The sign of the LoTable entry is used to choose one of two inner loops (for the "for" loop above): if LoTable>0, the inner loop contains TASK microprocessor functions; if LoTable<0, no TASKs are included (this is to aid in hardware debugging).

ProgramGeneratePage (directive=12). See section 1.4 for a complete description.

ProgramGenerateBand (directive=13). This function is very similar to **ProgramGeneratePage**, but will perform the image-generation chores for only *one* band, and leave the results in the Orbit *image* buffer: this function does *not* make arrangements to forward the image information to the ROS adapter. The intention is that the Alto program may want to read the generated image back into Alto memory rather than forwarding it to a ROS immediately.

The arguments to the function are as follows (see section 1.4 for a description of the data structures for fonts, left-overs and band lists):

LoTable = LoTablePointer (left-over pointer, must be even)
FontTable = FontTableX+100000b (pointer to font table, offset by 100000b)
NewCharPointer = BandListPointer-1 (BandListPointer must be even)

The function returns Result _ BandListPointer'-1, where BandListPointer' is a version of BandListPointer updated by processing all the entries for the one band (i.e., Result will point at the second word of the End-of-Band entry for the processed band). The idea is that **ProgramGenerateBand** can be called to generate the next band by setting NewCharPointer to the Result of the current band. (See also the description below concerning refreshing.)

The function **ProgramGenerateBand** makes no provision for reading the Orbit buffer back into the Alto. This can be accomplished with other **Program**xx functions.

ProgramROSStatus (directive=14). This operation reads an entire word (16 bits) of ROS status, using **OrbitControl** and **OrbitStatus** primitives. If we view the 256 status bits as lying in 16 16-bit words, numbered 0 to 15, then this function stores in Result the current value of status word *n*, where Argument1 = *n* * 1024. The firmware status bit unstableROSmc, reported in the OrbitStatus table entry, is zero if all 16 bits of status were extracted from the same status burst emanating from the adapter.

1.7 Refreshing the image buffer

As an image is being prepared in the image buffer, it may be necessary to refresh the buffer in order to keep its contents from changing (this is because the buffer memories are implemented with MOS RAM's). Reading an image from the output buffer not only clears the buffer but also refreshes it during the readout. When the reading completes, the buffers switch so that the output buffer becomes the image buffer. The image buffer must be refreshed by the microcode every 2 ms. until it is once again switched to become the output buffer.

The Orbit programmer must arrange so that the image buffer is refreshed if it contains valuable data. There are two cases to consider: (1) while a **Program**xx function is being executed, and (2) between invocations of **Program**xx functions. Most functions are speedy enough that no provision is made for refreshing. The **ProgramGenerateBand** and **ProgramGeneratePage** functions, on the other hand, automatically provide refreshing.

If the contents of the image buffer need to be preserved between invocations of primitive functions, it will be necessary to insure that the buffer is refreshed even while the Alto program is away doing other things (e.g., when the refreshing provided by **ProgramGenerateBand** is not functioning). To help with this requirement, the standard microcode has provision for performing the refresh duties *even when no **Program**xx function is being executed*. Thus, rather than turning off the Alto Orbit-controlling task between functions, the task remains active, waking up and refreshing the image buffer whenever it is appropriate. This *refresh idling* function can be specified by turning on bit 0 (100000b) in the Directive word of the control table, in addition to specifying a **Program**xx function

number in the directive. In this case, when the function terminates, the image buffer will continue to be refreshed. Enabling the idling activity in no way alters the invocation of the next **Program**xx function.

1.8 Reading a buffer into the Alto

Orbit has provisions for reading the *output* buffer back into Alto memory rather than shipping it to the ROS adapter. This is useful for for image-generation that is so complicated it cannot keep pace with the printing device: the buffers can be returned to the Alto, and buffered again on a big disk. A second pass is used to transmit the disk buffers (via Orbit) to the ROS. The read-back is also vital for debugging Orbit.

Basically, the function **OrbitOutputData** reads 16 bits from the output buffer into the Alto. The standard microcode interfaces **ProgramOutputData** and **ProgramReadBlock** are provided to give access to this function.

> The programmer needs to be aware of two tricky details concerning readout:
>
> 1. There is a 16-bit buffer (ROB) that sits between the addressing mechanism described above and the outside world (the taker of data: the Alto or the ROS adapter). As a result, immediately after the buffers "switch," the ROB contains the last 16-bits of data from the previous buffer (*x*=15, *y*=4080 through *y*=4095). The usual convention is to read out this *one last word*. This leaves the first word of the new buffer in ROB.
>
> 2. The *output* band buffer will not be *refreshed* because no readout is in progress. Because the buffer is implemented with dynamic MOS RAM's, the lack of refreshing will cause errors in the data to appear beginning about 2 ms. after the buffer switch. This data "rotting" happens rather slowly, but is nonetheless irksome. To avoid trouble, reading the output buffer should begin within the alotted 2 ms. after the buffer switch. As soon as reading begins, the reading itself serves to refresh the memory.

The scheme is as follows: (1) The Alto reads the output band buffer entirely (buffer A, say), thereby clearing it and also causing the buffers to switch; (2) Now the Alto invokes image-generation to fill the buffer (A) with an image; (3) The Alto reads the output buffer (B) as quickly as possible, simply to cause the buffers to switch; (4) The Alto reads the output buffer (A) and saves the image. Note that step 4 can also serve the same role as step 1. We could leave FA set to some fixed value throughout these steps, but this will cause step (3) to require more time than necessary. Some detailed analysis shows how to avoid this:

> The setting of FA provided by the Alto is examined only when Orbit reads out the *last* 16 bits of a scan-line, and must prepare to begin reading the next scan-line. If you wish to control FA carefully, you must anticipate the moments at which Orbit will load its address register from FA. The most common case in which careful synchronization is needed is when you wish to induce a buffer switch with minimum effort.
>
> Let us describe the handling of FA for the 4-step example above. Define FAimage to be the FA that corresponds to the image you wish to read. First, we need to get "in sync:"
>
> ```
> set Orbit's FA _ FAimage
> until buffers switch do [ read a word from the output band buffer ]
> read a final word from the output band buffer (because of ROB)
> ```
>
> This sequence guarantees FA has been set to FAimage, and that we are ready to begin reading a buffer. Step 1 (or step 4) becomes:
>
> ```
> read (256-FAimage)*16-2 words from the output band buffer
> set Orbit's FA _ 255
> read the last two words from the output band buffer
> ```
>
> Step 3 then becomes:
>
> ```
> read (256-255)*16-2 words from the output band buffer
> set Orbit's FA _ FAimage
> read the last two words from the output band buffer
> ```

1.9 Microcoding

This section presents some additional information on microcoding the Orbit. It refers generously to preceding sections, in which the basic data transfers (**Orbit**xx functions) have been described. The reader should be familiar with Alto microcode (see the Alto Hardware Manual), and may wish to refer to the Orbit functional description. The serious reader is urged to retain a copy of the standard microcode at his/her elbow as an example of how things can be done.

Assignment of functions. Each of the **Orbit**xx functions described above is assigned a value of the Alto F1 or F2 fields: Orbit decodes these fields to decide what to do. Definitions at the beginning of OrbitMc.Mu establish some conventional names for these functions. The assignment is as follows:

OrbitControl	F2=15b
OrbitStatus	F1=17b *
OrbitHeight	F2=12b *
OrbitXY	F2=11b
OrbitDBCWidthSet	F2=10b
OrbitFontData	F2=13b
OrbitDBCWidthRead	F1=15b
OrbitDeltaWC	F1=14b
OrbitROSCommand	F2=16b
OrbitOutputData	F1=16b
OrbitInk	F2=14b
OrbitBlock (see below)	F1=3

The starred items (*) may cause microcode branches. **OrbitStatus** sets NEXT[7] if IACS is *not* on, i.e., if Orbit is not in a character segment. **OrbitHeight** sets NEXT[7] if the refresh timer has expired, i.e. if the image buffer needs refreshing.

Whenever one of the microcode functions (**Orbit**xx) is executed, the Alto clock may not stop during its execution. The microcoder must arrange the microcode so that, for example, memory waits do not occur on cycles in which **Orbit**xx functions are executed. The standard microcode has comments (//// in the comment field) whenever instructions have been inserted to observe this timing restriction.

Wakeups. Orbit is controlled by a single Alto microtask (task number 1, next lowest priority to the emulator). Consequently, the Orbit hardware must generate a "wakeup signal" to govern the execution of microinstructions in the Orbit task: the wakeup should be generated when the Orbit hardware requires service of some sort.

The wakeup condition is very simply expressed in hardware, but deserves a more discursive treatment here. There are a number of considerations:

1. The RUN flip-flop. We want Orbit's wakeups to be filtered by a single RUN condition, that says whether Orbit is idle or is expected to be doing things. Whenever the Alto is reset (bootstrapped), the RUN condition is turned off. The emulator program may execute StartIO(4) to turn the RUN condition on (note that StartIO is simply the name of a software function that executes the emulator SIO function). The RUN condition is turned off by the execution of the **OrbitBlock** function *by the Orbit task itself.*

2. Executing miscellaneous functions (e.g., **OrbitROSCommand**). We would like the Orbit task to be normally active (i.e., requesting wakeup) so that various utility functions can be accomplished. When the function terminates, we can always deactivate the Orbit task with **OrbitBlock**.

3. Performing image generation. Again, we would like the Orbit task to be normally active, so that parameters and source raster data words can be shipped to Orbit at high speed. However, if Orbit's FIFO that contains font data words (shipped with **OrbitFontData**) becomes full, the Orbit task should be temporarily suspended until the FIFO entries are processed by Orbit and there is once again room. When image generation for a character segment terminates (IACS becomes 0), we want to get service no matter what the FIFO contains.

4. Waiting for the ROS to take a buffer. After image generation for a band is complete, Orbit usually needs to "wait" until the ROS has emptied the other (output) buffer. During this period, the Orbit task should not be requesting wakeups, so that the Alto microprocessor may be devoted to other tasks. To achieve this "dormant" state, the Orbit task sets the setGOAWAY bit in the control word: this tells Orbit to "go away" until a buffer switch occurs. The buffer switch will turn off GOAWAY, and consequently allow wakeups again.

5. Refreshing the image buffer periodically. If the image buffer memory is to be kept valid, it must be periodically refreshed. Consequently, we need to have a provision for waking up Orbit for refreshing even when Orbit is dormant (waiting for a buffer switch, as in (4) above). For this purpose, there is a REFRESH condition that indicates refreshing is needed: it is set every 2 ms. (approximately), and may be cleared by the CLRFRESH bit in the control word. Even though GOAWAY has been set, Orbit will be awakened when the REFRESH condition becomes true.

Let us summarize the conditions on which wakeups depend and give the wakeup condition itself:

RUN	Set by StartIO(4) Cleared by executing **OrbitBlock**
IACS	Set by executing **OrbitDBCWidthSet** Cleared when the character is finished (i.e., the width counter becomes 0 or the *x* counter overflows) Also cleared by executing StartIO(4) or by reseting Orbit
REFRESH	Set every 2 ms. by a timer Cleared by the CLRFRESH bit in **OrbitControl**
GOAWAY	Set by the setGOAWAY bit in **OrbitControl** Cleared by a buffer switch or by an Orbit reset

Orbit will request a wakeup if:

RUN and
{ [(FIFO has room) or IACS=0] and
[REFRESH=1 or GOAWAY=0] }

Refreshing. There is a standard ritual for refreshing the image buffer. Although the technique can be discerned from the addressing mechanism, we shall present it here for convenience. The basic idea is to cycle all the memories by generating a "character" of the right sort, but to leave the memory contents unaltered by using font data words that are zero. The suggested technique is (see the standard microcode for an instance of this technique):

OrbitXY _ 0
OrbitHeight _ 6000b (character is 1024. bits high)
OrbitDBCWidthSet _ 0 (character is 1 scan-line wide)
until IACS=0 do **OrbitFontData** _ 0

*1.10 Algorithm for **ProgramGenerateBand** and **ProgramGeneratePage***

The algorithm for the two central functions is best understood by consulting the text for the standard microcode. This section is simply a condensation of some of the information contained in comments in that file. In the description below, notation of the form a_b!c means that memory location b+c is read, and the 16 bits are transferred to destination a.

The microcode consists of three subroutines (DOBAND, TFRCHR, and REFRESH), and a main driver for the functions. DOBAND is responsible for sequencing through a band, calling TFRCHR (transfer a character to Orbit) for each character, and calling REFRESH every 2 ms. The driver for **ProgramGenerateBand** simply calls DOBAND and returns. The driver for **ProgramGeneratePage** is somewhat more complicated:

```
    OrbitControl _ 300b (turn on SLOTTAKE)
    OrbitControl _ FAword + 4 (WHICH, plus the FA setting)
    bandResult _ nBandsMinus1+1

Main loop.  First execute any pending adapter command entries.
    while nBandsMinus1 > 0 do begin
        while (Argument1!0-nBandsMinus1) and 7777b = 0 do begin
            a_(Argument1!0 and 30000b)
            b_Argument1!1
            Argument1 _ Argument1+2
            if a = 0 then OrbitROSCommand _ b else
            if a = 10000b then begin
                OrbitControl _ b
                Argument1!-1 _ OrbitStatus
                end else
            if a = 30000b then wait b/3 microseconds
        end

Now call the DOBAND subroutine to compose the image for this band.
        call DOBAND
        OrbitControl _ 10b (set GOAWAY)
        if AdapterStatus.SendVideo = 0 and OrbitStatus.BEHIND = 1 then
            return from ProgramGeneratePage with earlyPageAbortmc bit set
        if AdapterStatus.(word 8, bit 3) = 1 then bandResult _ nBandsMinus1
        a _ Argument2
        until bands switch do begin
            wait until refreshing needed or bands switch
            if a < 0 then return from ProgramGeneratePage with TIMEOUTmc bit set
            call REFRESH
            a _ a-1
        end
    nBandsMinus1 _ nBandsMinus1-1
    end
    Result _ bandResult
    return
```

The DOBAND subroutine is responsible for sequencing through the band lists and leftovers:

```
DOBAND:     LORP_LoTable-1
    LOWP_LoTable-1          (LOWP is the write pointer for new leftovers)

First, process the left-over list remaining from last time.  Each entry is four words long, containing the height, the
XY setting (with X=0), the remaining width and the memory address of the first source raster word to send to
Orbit.  Note that the height and XY information is also saved in registers (HEIGHT, XY) so that TFRCHR can
build a new left-over list:
    while LORP!1 = 0 do begin
        if refreshing needed then call REFRESH
        OrbitHeight _ HEIGHT _ LORP!1
        OrbitXY _ XY _ LORP!2
        OrbitDBCWidthSet _ LORP!3
        FONTADR _ LORP!4
        call TFRCHR
        LORP _ LORP+4
    end
```

Now go through the new character list (band list) for this band, and decode the entries. Character entries are looked up in the font index; rule parameters are simply passed on to Orbit. (Note that the "refreshing needed" test is a part of **OrbitHeight**, and is consequently done in a slightly different place in the actual microcode.)

```
    while true do begin
        if refreshing needed then call REFRESH
        a _ NewCharPointer!1
        XY _ NewCharPointer!2
        NewCharPointer _ NewCharPointer+2
        if a < 0 then begin
            OrbitXY _ XY
            FONTADR _ FontTable!a
            OrbitHeight _ HEIGHT _ FONTADR!0
            OrbitDBCWidthSet _ FONTADR!1
            FONTADR _ FONTADR+2
        end else case a&#37 of begin
            case 1: begin
                OrbitXY _ XY
                FONTADR _ 0
                OrbitHeight _ HEIGHT _ NewCharPointer!1
                OrbitDBCWidthSet _ NewCharPointer!2
                NewCharPointer _ NewCharPointer+2
                end
            case 4: begin
                NewCharPointer_NewCharPonter+XY;
                repeat the loop
                end
            case 0: exit the loop
            all other cases: return from ProgramGeneratePage with badBandEntrymc bit set
        end
        call TFRCHR
    end

    LOWP!1 _ 0
```

The TFRCHR subroutine transfers a character to Orbit. If the character does not exhaust its width in this band, a left-over entry is made. The variable FONTADR usually contains the address of a source font word, but if it is zero, we are transferring a "rule" to Orbit.

```
TFRCHR:     if FONTADR=0 then begin
            until IACS=0 do OrbitFont _ -1
            FONTADR _ - OrbitDeltaWC
    end else begin
            p _ FONTADR
            until IACS=0 do begin
                OrbitFont _ p!0
                p _ p+1
            end
    end
```

Now check to see if the charcter requires a left-over (width=7777b):

```
    p _ OrbitDBCWidthRead and 7777b
    if p = 7777b then begin
        LOWP!1 _ HEIGHT
        LOWP!2 _ XY and 7777b (this sets X_0, but preserves Y)
        LOWP!3 _ OrbitDBCWidthRead
        LOWP!4 _ FONTADR + OrbitDeltaWC
        LOWP _ LOWP+4
    end
```

And finally, the REFRESH subroutine performs the refreshing chore:

```
REFRESH: OrbitXY _ 0
    OrbitHeight _ 6000b
    OrbitDBCWidthSet _ 0
    until IACS=0 do OrbitFont _ 0
```

2. ROS Adapter

The ROS adapter is a general-purpose interface between EIP's (Electronic Image Processors; Orbit is one) and raster-scanned output devices. The motivation underlying its design is that one adapter can serve to connect the "9-wire ROS interface standard" to a number of similar ROS devices. There are two slightly different versions of the adapter. Dover systems come with a "TTL adapter," hereafter abbreviated T. A high-speed version of the adapter has been built by SDD using MECL 10K logic, called the "MECL adapter," hereafter abbreviated M. In mid-1978, the T adapter was redesigned, with a few minor "visible" effects. This revision will be referred to as the "Dover II" adapter.

The adapter contains logic to control fully the raster output scanner (ROS) itself, including of course generating a video signal. It also contains a mechanism to receive printer commands from the EIP and forward them to the printing engine, as well as a mechanism for receiving engine status and forwarding it to the EIP. (The standard Orbit microcode provides access to the adapter with the **ProgramROSCommand** and **ProgramROSStatus** functions.)

2.1 Basic imaging technique

The EIP delivers raster images of pages to the adapter, which is responsible for formatting it properly on the output device, and for timing signals properly. The adapter is basically a massive synchronization aid: it receives timing information from the printing engine and the ROS equipment, and generates requests for video data (to the EIP) in order to keep up. The video data it receives must be converted into a serial video signal, timed properly with respect to the scanning motion, so that the image appears in the proper location on the "page."

A simple conceptual model of the imaging process will help in the discussion of the rather intricate chores of the adapter. Although the model is somewhat oversimplified, it includes the basic ideas common to all the devices that the adapter can drive. Figure 2-1 shows the model schematically: there is a single scanning beam that scans in a fixed plane, under which a continuous sequence of "page image frames" is being moved. If we properly time the modulation of the scanning beam, we will leave behind within each page image frame an image that corresponds to the raster prepared by the EIP. A number of timing problems must be solved for this to happen properly.

Engine operation. The printing engine reports a signal PrintMode to the adapter whenever its drive motors are running: this indicates that "page motion" in our conceptual model is in progress, even though pages may not be flowing under the ROS beam yet.

Page timing. We will need to know when the leading edge of a page image frame passes under the scanning beam. It is at this instant that we must begin acquiring raster data from the EIP and using it to modulate the scanning beam. This timing information can only be acquired with the help of the printing engine itself, for it is responsible for sequencing pages. The engine generates a signal called PageSync some fixed time before the page frame passes under the beam. Once inside the adapter, this signal is delayed to generate a timing signal called SendVideo which is true while a page image frame lies under the beam, and false in the inter-page gaps. The adapter will process EIP raster data only when SendVideo is true.

> On the M adapter, SendVideo is named "DelayedPageSync" for historical reasons.
>
> We mentioned above that Orbit needs to write the first few bands of raster information in a spot that will not be visible on the page, as they may contain garbage that has accumulated because refreshing has not occurred. In this case, Orbit should arrange to have SendVideo occur slightly before the actual page frame would pass under the beam.

The normal sequence of events, then, is roughly as follows: (1) The EIP issues commands to the printing engine to start pages feeding. (2) The EIP waits for SendVideo to disappear (the reason for this will appear later). (3) The EIP resets the adapter buffer mechanism and starts

generating raster data. (4) When SendVideo becomes true as the first page frame passes under the beam, the adapter will begin requesting raster data from the EIP, and continue to do so as needed to control the beam modulation. Now one of two things can happen: (5a) SendVideo falls, signaling the end of the page frame. The adapter will cease requesting data from the EIP. Or (5b) the EIP refuses to honor a data request from the adapter, in which case the adapter will simply repeat the last data it received until SendVideo goes false. (6) To print the next page, the EIP returns to step (2), and may also need to contract with the printing engine to keep feeding paper. The reason for step 2 now becomes clear: without step 2, situation 5b might cause video for the next page to appear at the trailing end of the current page, because SendVideo is still on.

Scan-line timing. An even more intricate timing problem is posed by the scanning beam: we must carefully synchronize the video signal generated by serializing the raster data coming from the EIP so that it appears at the proper position along the scan-line. This involves two issues: (a) determining how to divide the scan-line into units that will correspond to each video *bit*, and (b) determining the absolute time at which video bits should be passed to the beam modulator.

A servo system and two detectors are used to solve this problem. The scanning beam passes first over a "start-of-scan" (SOS) detector, then over the page image frame, and then over an "end-of-scan" (EOS) detector. This sequence is repeated very fast: soon after EOS is generated for one scan-line, SOS for the next scan line occurs. The adapter divides the *time* between SOS and EOS into equal intervals (the exact number of intervals is a parameter set by the EIP): each one will correspond to one bit of video information. So now we have divided the scan-line into "bits."

The EIP generally produces raster data only for the portions of scan-lines that correspond to regions of activity on the page: the adapter is expected to supply "white" margins above and below this active region. Consequently, the adapter will need to know how the video data supplied by the EIP should be placed within the scan-line. The adapter implements a "bottom margin" by a counter that is started at SOS with a value specified by the EIP, and counted as "white margin" bits are sent to the modulator. When the count becomes 0, video data from the EIP is sent to the modulator, and continues to be sent until an "end of scan-line" indication arrives with data from the EIP. Thereafter, the adapter supplies "white margin" bits that will correspond to the "top margin."

Note: The number of bits of raster data provided by the EIP for each scan-line must be a multiple of 64. (With Orbit, this means that FA must be a multiple of 4.)

> The servo that synchronizes scan-lines requires SOS and EOS pulses to arrive. Consequently, the adapter control must arrange that the laser beam is "on" as it passes over the detectors, or a pulse will be missed. When the servo is operating in equilibrium, the adapter ensures that the beam will be on for the detectors by anticipating where the beam is. However, in order to get the servo going initially, the beam must be completely on for a short while. This is accomplished by turning the beam on whenever PrintMode is true but SendVideo is not, i.e., in the inter-page gaps and while the engine is cycling up. This has the added virtue of "writing down" the photoreceptor in unused areas, thus preventing toner dumping.

Motor speed. The speed of the motor that drives the scanning polygon is carefully controlled by a frequency synthesizer. The adapter provides the basic timing signal to which the electronics locks the polygon motor.

2.2 Parameters

All of these timing and synchronizing features are governed by parameters that can be set by the EIP. Some of the parameters can be related by physical observations. The next few paragraphs are intended to provide enough information to permit calculation of the parameters!

Let us first define some parameters of the output image:

S	The number of scan-lines per inch on the page
B	The number of bits per inch along the scan-line

Now define some parameters of the ROS and printing engine (see section 3 for the values applicable to the Dover printer):

p	Paper speed, measured in inches per second
f	Number of facets on the polygon
r	Number of polygon motor clock pulses/polygon revolution
d	Duty cycle (i.e., SOS-EOS time/SOS-SOS time)
h	Effective distance between SOS and EOS detectors, measured at the paper surface.

From these we will need to calculate the relevant adapter parameters (ranges are given in parentheses):

MotorSpeed (0 to 4095)
MotorScale (0 to 7)
BitClock (0 to 4095)
BitScale (0 to 7)
LineSyncDelay (0 to 4095)
PageSyncDelay (0 to 4095)
VideoGate (0 to 4095 -- T version only)

We need to define two parameters that differ on the different adapter versions.. The first is bcMax, the maximum frequency of the bit clock. The other is crystalClock, the frequency of an internal adapter timing oscillator.

	M version	T version
bcMax	$90 * 10^6$	$30 * 10^6$
crystalClock	$25 * 10^6$	$12.5 * 10^6$

Define a function $Scale(x)=2^x$.

Now we can calculate the scanning motor speed in revolutions per second:

$$MotorRPS = crystalClock * Scale(MotorScale) / (r * (4096-MotorSpeed) * 2^8)$$

Consequently, we can get the number of scan-lines per inch:

$$S = f * MotorRPS / p$$

These two equations allow us to determine MotorSpeed and MotorScale settings from the desired value of *S*.

The scan-line control is somewhat more complicated. The basic servo is controlled by the relation:

$$B = 4 * (4096-BitClock) / h$$

The peak bandwidth required from the adapter (and consequently from the EIP as well) is:

$$BitRate = f * MotorRPS * B * h / d$$

bits per second. (If Orbit is driving the adapter, it can deliver at most $23 * 10^6$ video bits per second.)

An important side condition governs the setting of BitScale so that the bit clock servo operates in a reasonable range:

$$1/2 < 2^7 * \text{BitRate} / (\text{bcMax} * \text{Scale}(\text{BitScale})) < 1$$

The remaining parameters are more easily calculated. LineSyncDelay is simply (4096-*n*/4), where *n* is the number of bits of white margin to leave at the bottom of the page (after the SOS detector). PageSyncDelay is (4096-*n*/*i*), where *n* is the number of scan-lines to pass up after receiving PageSync from the engine before starting SendVideo, and *i* is 1 for Dover II adapters, 4 for older ones. And VideoGate is (4096-*n*/4), where *n* is the number of scan-lines to pass up after SendVideo starts before stopping SendVideo (on the M adapter, VideoGate is not provided, and the falling of SendVideo is controlled by the printing engine).

2.3 Adapter commands

Control of the adapter, ROS and printer is accomplished with 16-bit commands. The high-order 4 bits of the command give a *command code*; the remaining 12 bits are used as an argument to the command. The command codes are:

0 Buffer reset.

1 Set scales.
BitScale _ command[4-6]
MotorScale _ command[7-9]
ExtendVideo _ command[10] (T version only)
TestPageSync' _ command[12] (should normally be =1)
CommandLocal _ command[13]
CommandBeamOn _ command[14]
TestMode _ command[15]

2 Set bit clock register.
BitClock _ command[4-15]

3 Set motor speed register.
MotorSpeed _ command[4-15]

4 Set line sync delay register.
LineSyncDelay _ command[4-15]

5 Set page sync delay register.
PageSyncDelay _ command[4-15]

6 External command 1 (forwarded to printing engine)
ExternalCommand1 _ command[4-15] (a register is set)

7 External command 2 (M version only)
ExternalCommand2 _ command[4-15] (a register is set)
Set video gate (T version only)
VideoGate _ command[4-15]

10b-17b Spare

2.4 Adapter status

The adapter constantly reports 256 status bits to the EIP. These bits are normally viewed as consisting of 16 16-bit words. The first 8 words of status are reasonably independent of the kind of adapter (T or M) or the exact kind of printer attached to the adapter:

Word	Bits	Function
0		Special status from the ROS
	0	SendVideo (sometimes called DelayedPageSync)
	1	PrintMode
	2	Local
	3	BeamEnable
	4	StatusBeamOn
	5	StatusPowerEnable (M version only)
1		Command register
	0-15	A copy of the command most recently received by the adapter
2		Bit clock
	0	VideoPolarity (the setting of a switch)
	1-3	BitScale (register set with command code 1)
	4-15	BitClock (register set with command code 2)
3		Motor speed
	0	SelectLeadEdge (the setting of a switch)
	1-3	MotorScale (register set with command code 1)
	4-15	MotorSpeed (register set with command code 3)
4		Line sync delay
	0	Switch3 (the setting of a switch)
	2	ExtendVideo (register set with command code 1 -- T version only)
	3	TestPageSync' (register set with command code 1)
	4-15	LineSyncDelay (register set with command code 4)
5		Page sync delay
	0	Switch4 (the setting of a switch)
	1	CommandLocal (register set with command code 1)
	2	CommandBeamOn (register set with command code 1)
	3	TestMode (register set with command code 1)
	4-15	PageSyncDelay (register set with command code 5)
6		External command 1
	0	LineNoise
	1	CompareError
	2	BufferUnderflow
	3	PacketsOK
	4-12	ExternalCommand1 (register set with command code 6)
7		
	0-3	LineCount
	4-15	ExternalCommand2 (register set with command code 7; M version only)
	4-15	VideoGate (register set with command code 7; T version only)

The remaining 8 words of status are normally used for external (engine) status of various kinds. The TTL adapter organizes these words as follows:

Word	Bits	Function
8	0-15	Special status bits 0-15 (see Dover section for interpretation)

9	0-15	Special status bits 16-31 (see Dover section for interpretation)
10	0-15	ID (16 bits). This identifies the engine type.
11	0-15	Serial number (16 bits). This specifies the serial number of the engine.
12-15	0-15	Additional engine-dependent values, used by some printers (not Dover).

2.5 Additional adapter features

There are a number of additional adapter features, chiefly for providing various kinds of diagnostic and debugging help.

The ExtendVideo flag (T version only) can be used to override the action of the VideoGate counter. Once SendVideo comes on, it will stay on until ExtendVideo is turned off *and then* the VideoGate counter reaches zero.

There are two variations of "local" operation for checking out the printing engine. The machine may be switched to Local by a switch on the engine (and reported as a status bit), or may be set into this mode by setting CommandLocal. In the first case, the adapter extracts commands from a PROM that are intended to start paper motion and printing. Two switches (Switch3 and Switch4) govern the selection of one of four local command sequences that is extracted from the PROM. In both local cases, the adapter will generate "graph paper" video signals governed by the video gate counter (VideoGate) and the line sync delay register (LineSyncDelay). In order to see graph paper cover the page, VideoGate and LineSyncDelay must be set in such a way that they are counting as the beam passes all spots on the page.

To aid debugging the adapter itself, the TestMode bit may be set. This permits the crucial printer and ROS timing signals to be generated in the adapter rather than in the engine. If TestMode is set, PageSync is taken from the *complement* of the control bit TestPageSync', and line sync (analogous to start-of-scan) is generated by the motor control circuitry: on the M adapter, it will be on for (15/16) * (4096-MotorSpeed) / crystalClock seconds and off for (1/16) * (4096-MotorSpeed) / crystalClock seconds; on the T adapter, it will be a signal on for 4 * (4096-MotorSpeed) / crystalClock seconds and off for the same amount of time. The "on" portions of these signals simulate scan-lines of the corresponding durations.

The BeamEnable status bit means that the doors to the ROS housing and/or printing engine are closed, and the interlocks have engaged.

To force the laser beam on, and consequently to allow the servos to settle down, either CommandBeamOn or StatusBeamOn may be set. CommandBeamOn may be set by the EIP, and StatusBeamOn can be set with a switch in the ROS housing (M adapter).

There are four switches on the adapter module that can be used to alter the operation slightly. The VideoPolarity switch will invert the sense of the binary signal sent to the beam modulator. SelectLeadEdge is a switch that selects either leading or trailing edges of the SOS and EOS signals for careful timing purposes (different ROS boxes operate differently).

Adapter errors. The adapter keeps track of various kinds of errors that may occur during its operation, and reports some status bits. If the adapter detects errors on the line that delivers commands to the adapter, it sets LineNoise or CompareError and ignores the command. If the EIP fails to provide video data fast enough, the adapter sets BufferUnderflow (this condition is reset by the Buffer reset command). The PacketsOK condition is set by the Buffer reset command, and cleared whenever an improperly-formatted video data packet is received from the EIP.

3. The Dover Printer

The Dover printer is a Xerox 7000 copier, modified to substitute a ROS module for the optics and to incorporate an *engine controller* that permits the printing operations to be controlled adequately by the adapter. This section describes the standard Dover engine, which operates at a paper speed of 10 inches/second. Dover can be "extended" in an experimental setting to run at 5 inches/second. The timing for the extended Dover is *very* different than the timing described here.

3.1 Engine Parameters

Several engine parameters were mentioned in the adapter section. The values for Dover are:

p	10 inches/second
f	32 facet polygon
r	24 clocks/revolution
d	.90 scan-line duty cycle
h	12.5 inches

Typical settings in the T adapter for *S*=*B*=350 bits/inch are:

BitRate=17 * 10^6
MotorRPS=109.38
MotorSpeed=1707
MotorScale=7
BitClock=3002
BitScale=7
LineSyncDelay=4046 (3008 for entire line for graph paper)
PageSyncDelay=3971 (3596 on Dover II)
VideoGate=3352

3.2 Engine timing

The engine timing for Dover is somewhat intricate, as there is a substantial "pipeline" effect in the paper path. The philosophy used in designing the engine control electronics has been to make them simple, and to require the EIP program controlling the engine responsible for sorting out most of the timing details.

There is only one signal used to instruct the Dover engine: a PrintRequest signal that is generated by a 0-to-1 transition of the low order bit of ExternalCommand1. Thus two successive adapter commands (first 60001b, and then 60000b) are normally used to cause the PrintRequest signal. The PrintRequest signal is used by Dover to feed sheets; be warned, however, that initiating and terminating a printing sequence are both a bit tricky, and require careful thinking (more on this below).

Dover generates only one timing signal of interest: CS-5, which is transmitted to the adapter as the PageSync signal. All timing information relevant for paper-path motion is derived from this signal. For purposes of discussion, it is helpful to "number" each of the CS-5 pulses generated by the engine, starting with CS-5(0), the first to be generated as the machine cycles up.

We shall describe the operation of Dover by describing the various sequences involved: (a) a cold start assuming the motor is presently off (PrintMode is off); (b) the "inner loop," in which paper is happily flowing through the engine, and page after page is being printed; (c) the runout shut-down sequence; and (d) the malfunction shut-down sequence.

Inner loop. We shall begin by describing the inner loop, as it is the simplest sequence. Refer to Figure 3-1 for an illustration of the timing. Let us assume that CS-5(n) has just occurred. Approximately 250 ms. later, the adapter should begin sending video to the ROS that will correspond to the leading edge (left-hand edge) of the paper. This time will vary a little from machine to machine, and can be controlled with the help of the PageSyncDelay register in the adapter. Imaging the page persists for about 850 ms. Approximately 896 ms. after CS-5(n), the Count-H status signal is generated and persists until the next CS-5 (Emperically, Count-H is on for only about 20ms. in older adapters; it is much wider in Dover II adapters. Special provisions in the standard microcode provide help in detecting this signal reliably.) This signal indicates that paper was successfully fed to hold the image for the page that is being imaged on the drum. If Count-H does not appear at the proper time, it is likely that some malfunction has, or is about to, occur. Within 990 ms. after CS-5(n), it is necessary to issue a new PrintRequest (i.e., to send the adapter a "set External Command 1" command) if another sheet is to be fed (i.e., if you desire to keep printing at high speed).

Cold start. In order to initiate printing, the EIP issues a PrintRequest (again, by issuing the "set External Command 1" command to the adapter). PrintMode should come on, verifying that power has been applied to the main motor. About 250 ms. after the print request, CS-5(0) is generated. This first CS-5 identifies a machine cycle that will *not* result in an output page: if you were to interpret CS-5(0) in the fashion described above for the inner loop, the first image you delivered would not be blessed with a sheet of paper to receive it. However, if you issue a second PrintRequest within 990 ms. of CS-5(0), the machine will cycle again, and generate CS-5(1). *This* CS-5 does in fact correspond to a sheet of paper -- now you may enter the inner loop.

A convenient way to think of the cold start sequence is to start the inner loop with CS-5(0), with two additional features on the first page imaged: (1) it will not be transferred to paper, and should therefore be "white" (in order to insure the bit clock servo is running properly), and (2) the Count-H signal will not be generated, because no sheet of paper was actually fed.

Runout shut-down. Dover begins shut-down whenever it fails to receive a PrintRequest in time (i.e., within 990 ms. of a CS-5). The machine must continue to operate for some time, however, in order to allow the last sheet of paper to be fused and transported to the output hopper. There will be 7 gratutitous CS-5 pulses generated after the last CS-5 that was produced by a PrintRequest. At the end of this sequence, PrintMode is turned off, indicating that paper path motion has stopped. In order to re-start the machine, a cold-start sequence is required.

If you wish to resume printing before the 7th CS-5 has passed, you may resume issuing PrintRequests, just as if you were in the inner loop (i.e., the first CS-5 after your PrintRequest will be blessed with a corresponding sheet of paper).

Malfunction shut-down. When a malfunction is detected, the Dover printer shuts down immediately, and does not wait for paper to be transported out of the machine. An appropriate malfunction status bit will be turned on (see next section), and the machine will halt (PrintMode goes away; no more CS-5's will happen). After an operator has corrected the problem, the machine must be restarted with the cold-start sequence.

3.3 Engine status indications

Dover may report up to 32 bits of status to the EIP via the adapter. Most of these bits are unused. The table below gives the name of each status bit and a short description of its purpose. The various malfunction indications are followed by a quoted string in italics; this is the recommended operator message for the condition (adherence to standard messages simplifies the job of the trouble-shooter).

Word	Bit	Signal
8	3	Count-H. This signal is raised if a sheet has been successfully fed to receive the image for the page being imaged. Count-H comes on 896 ms. after CS-5 and persists until the next CS-5.
8	5	PTDisorder. This signal is active when the paper tray is not in the up position or when the paper tray cover is open. It persists until the condition is corrected. This is the "or" of (not LS4) and (not LS24&LS31). *"Paper tray open."*
8	8	* LS4. This signal is active when there is adequate paper in the paper tray. A zero value will also assert PTDisorder.
8	10	LS27. This signal is active when progress from the A transport to the register stop module is not normal. It persists until the paper is cleared. *"Jam--paper feed."*
8	11	* LaserOn. This signal is active when the laser power supply is on, and the beam is available for use. *"Laser is off."* (Applies when the status bit is *not* present.)
8	12	* LS22. Malfunction Reset. Left as an exercise to the reader.
8	13	ReadyTemp. This signal is active when the fuser temperature is above 285 degrees F. This corresponds to the minimum temperature needed to fuse the toner to the paper. *"Fuser not warm."* (Applies when the status bit is *not* present.)
8	14	LostPower. This signal is active when machine power is turned off when PrintMode was active (i.e., the main drive motor was energized). It persists until the paper is cleared. *"Lost engine power."*
8	15	* ModeCont. This signal is active if the machine is set to run in its "extended", or half-speed, configuration.
9	1	PhotoCellOut. This signal is active when a sheet of paper has not been knocked off the drum during machine operation. It persists until the paper is cleared. *"Jam--paper on drum."*
9	2	PreSeq. This signal is active if a malfunction occurs before the first page is imaged (i.e., during the power-up sequence). *"Jam--startup sequence."*
9	4	* LS9. Two pieces of paper were fed. "*Jam--two sheets fed."*
9	5	ACMonitor. This signal is active when the machine is powered up. It is possible to have ReadyTemp-H active even though the ACMonitor-H is not. *"Engine not powered up."* (Applies when status bit is *not* present.)
9	6	LS38. This signal is active when a sheet of paper is jammed on the fuser roll. *"Jam--paper on fuser roll."*
9	8	* LS1. B Transport Jam. *"Jam--B Transport."*
9	9	Malfunction. This is a general-purpose signal that is the "or" of all the possible error indications given above (PTDisorder, LS27, LS22, LostPower, PhotoCellOut, PreSeq, LS38, LS1, LS3).
9	10	LS3. This signal is active when a sheet is not knocked off the drum and PhotoCellOut-H does not catch it. It persists until the paper is cleared. *"Jam--paper on drum."*
9	13	* LS24&LS31. Paper tray is up and sensing bar is in place. Normally active. A zero value will also assert PT-Disorder.

* Meaningful for Dover II adapters only.

4. Performance

This section gives preliminary results on the performance of Orbit in actual printing runs. The basic test is to place as many characters of a given size on a page as possible before Orbit "gets behind." Orbit will get behind when the demands of composing video for a complex page exceed the speed capacities of the Alto and Orbit.

For these tests, Orbit is attached to a Dover printer running at 10 inches per second, 350 scan-lines per inch, 350 bits per inch. The Alto II driving Orbit is perfectly standard--the "disk" it uses is a Diablo Model 31. The fonts used are all versions of Helvetica, scan-converted from spline representations. The tests reported here use the "standard" Orbit microcode, and do not resort to trickery of any kind. Orbit is flexible enough to do quite a bit more than is reported here.

Bandwidth Capacity

Character point size	Nominal chars/page	With disk	Without disk
Portrait:			
6	>11632*	>11632*	>11632*
8	8748	>10260	11137
10	5460	>7560	7980
12	3618	>6300	<6365
14	2622	>4503	4731
Landscape:			
6	>14000*	>14000*	>14000*
8	8576	>11008	<11124
10	5508	>8214	8724<9030
12	3735	>5058	>6516
14	2652	>4900	>5035

Explanation. The nominal number of characters per page is the number of characters of the given size that fit comfortably (without squeezing or overprinting) on a page. These numbers compare reasonably well with typical pages printed on EARS. The third and fourth columns give Orbit's measured capacity. The third column is measured with disk activity present (the assumption is that while printing a page of a given complexity, you must be reading from the disk into another buffer a description of the next page, so that printing at the given capacity can continue uninterrupted). The fourth column is measured with disk activity absent. Note that absence of disk activity increases capacity only slightly. (Starred items exceeded character sort size in my test program.)

Storage capacity

Character point size	Font storage in words/character
Portrait:	
6	24.3
8	39.8
10	59.1
12	83.7
14	111.
Landscape:	
6	23.9
8	39.4
10	58.6
12	82.9
14	111.

The table above gives storage requirements for a single character, including the necessary index table. Note that with Orbit, it is straightforward to economize on font storage by including only those individual characters known to be used on the page.

Storage requirements for a "page description" (i.e., information about which characters are to appear where) are:

2 words/character +
4 words/rule (horizontal and vertical lines) +
st/8

where st is the total number of scan-lines on the (active) part of the page.

Rule of thumb

There is a model (the details of which I will avoid stating here) that can be used to estimate the numbers given above. Let *r* be the resolution of the ROS in bits/inch. Let *s* be the point-size of characters. Let *p* be 1.0 for fixed-pitch fonts and .6 for proportionally-spaced fonts. Let *t* be the total imaging time in seconds (.85 for Dover).

The number of font storage words per character is roughly $3 + 7.2*10^{-6}\ p\ (rs)^2$. The maximum capacity of Orbit in characters per page is roughly $2.1*10^{6}\ t/(2.1*10^{-5}\ p\ (rs)^2 + 2.3*10^{-2}\ p\ rs - 6)$.

5. Miscellaneous

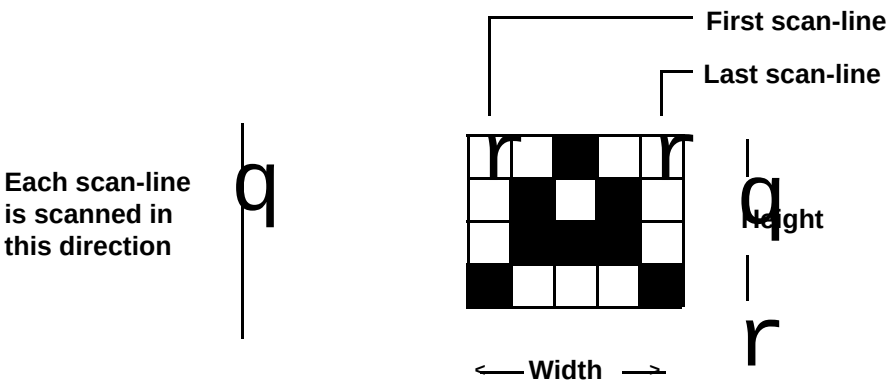
A pattern generator for gray (shaded) areas. A simple algorithm will generate acceptable gray patches on most printers. The scheme, devised by Mike Wilmer, is used to compute an 8-by-8 bit pattern that can be replicated laterally to produce the shade (the INK memory can help with this). If you want a darkness *D*, where 0 corresponds to white, and 63 to black, a bit of the pattern should be 1 (black) if *D* is greater than the entry in the following 8-by-8 table:

44 39 31 17 09 25 37 52
20 26 33 41 49 35 28 12
00 10 50 57 61 46 22 02
06 18 42 58 62 54 14 04
08 24 36 53 45 38 30 16
48 34 29 13 21 27 32 40
60 47 23 03 01 11 51 56
62 55 15 05 07 19 43 59

References

Severo Ornstein, "Orbit General Description," PARC/CSL Memo.
Orbit Logic Drawings
Orbit Standard Microcode, saved on <DOVER>OrbitMc.Mu

"ROS Adapter Functional Description," saved on RosAdFuncDesc.Ears.
MECL version logic drawings saved on RosAdFiles.dm.



Raster bit stream:

1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1 0 0 0

Raster bit stream, packed into 16-bit words:

1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0

1 0 0 0 (padded with zeroes)

Figure 1-1: Source raster and encoding

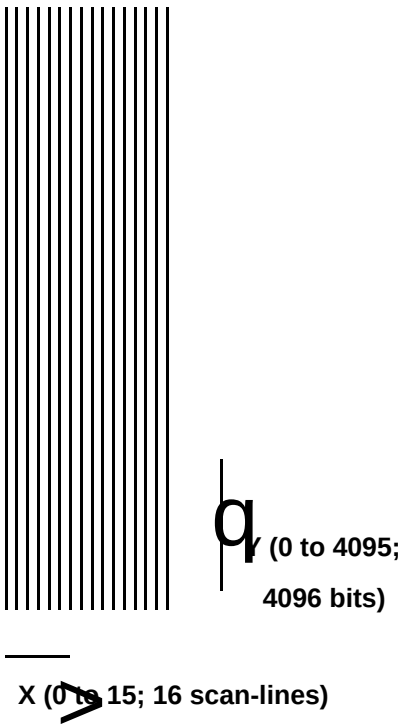


Figure 1-2: Band buffer

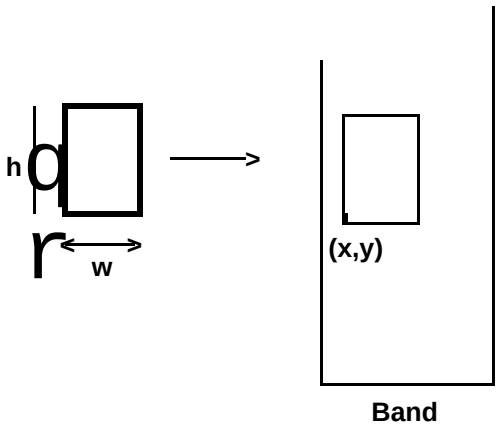
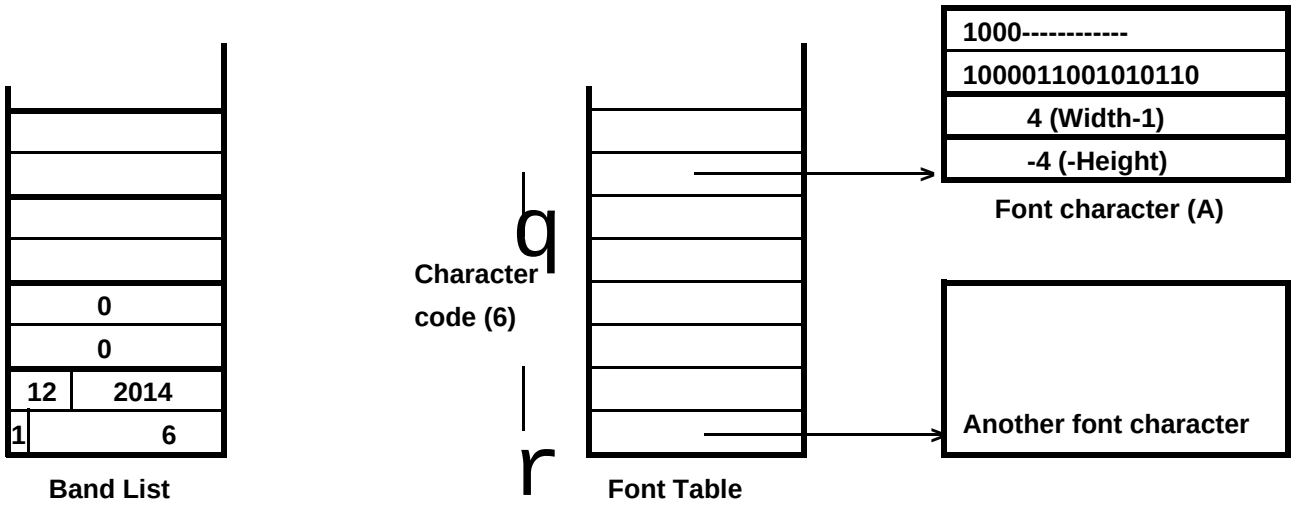


Figure 1-3: Copying a source raster



The band list shows a single character (code=6, x=12, y=2014), followed by an end-of-band indicator. Memory addresses increase upward.

Figure 1-4: Page generation data structures

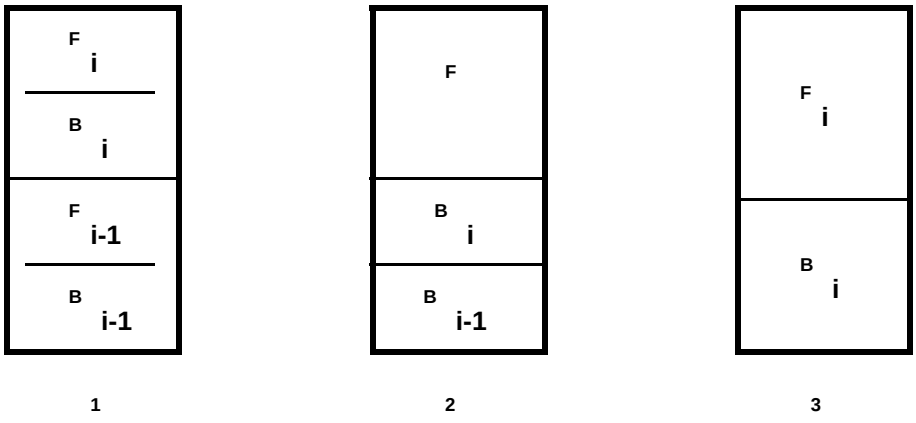


Figure 1-5: Alternative memory-management schemes

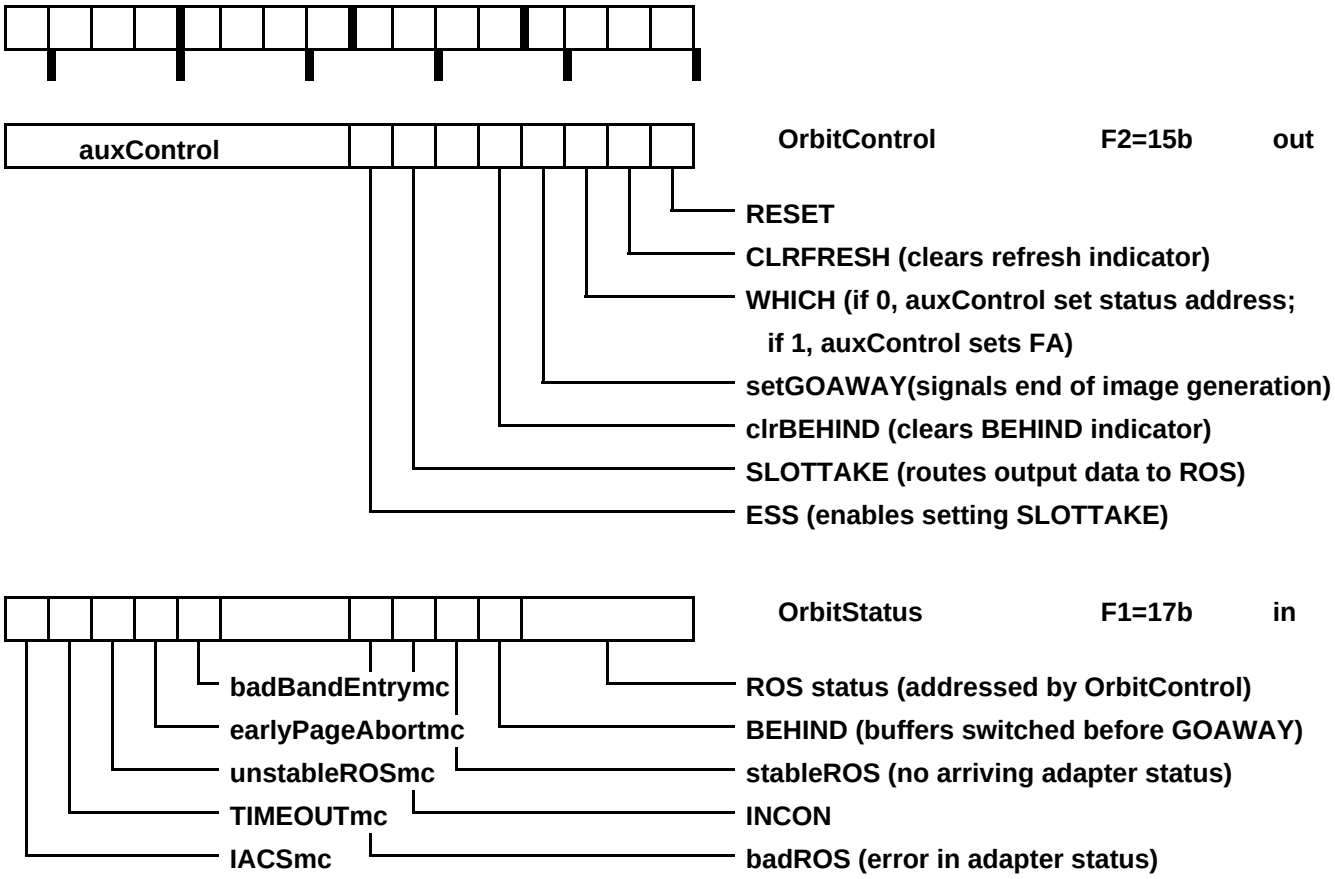


Figure 1-6: Orbit functions

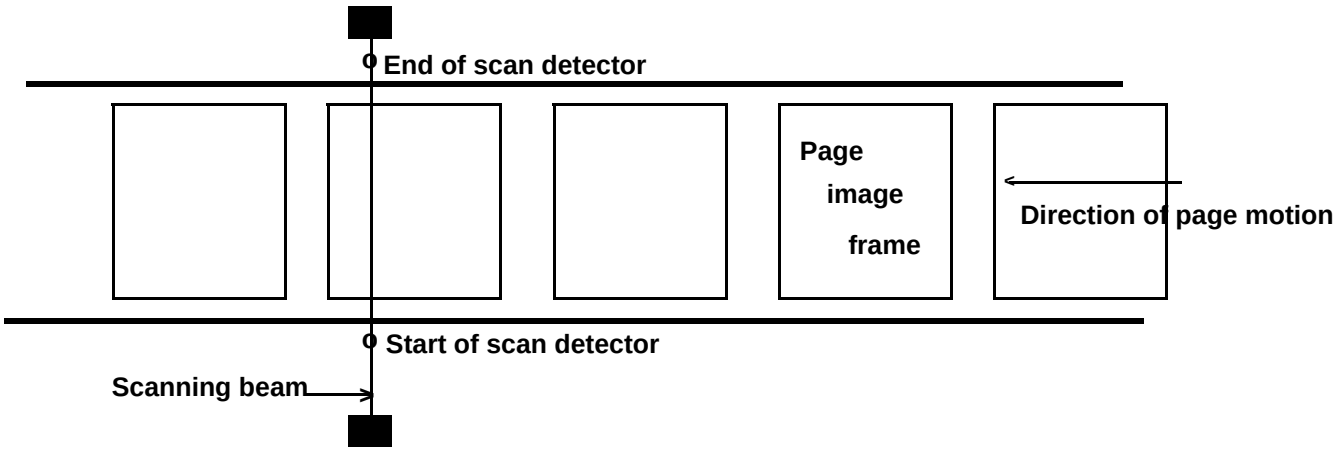
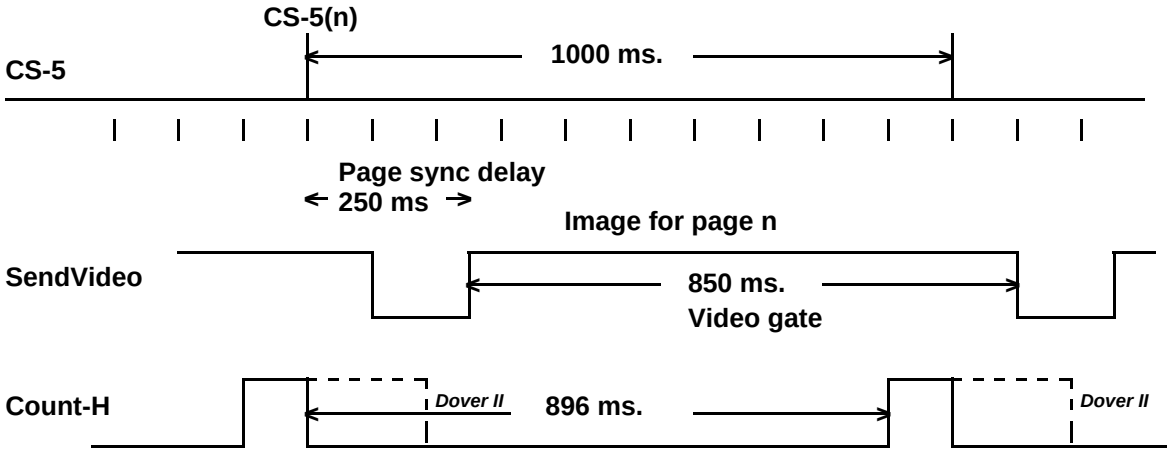


Figure 2-1: Conceptual model of the imaging process



SendVideo is generated in the adapter by counting PageSyncDelay after CS-5 arrives, and then counting VideoGate before terminating SendVideo. All counting is done in terms of scan-lines.

Figure 3-1: Dover printer timing