Array of Intensity Samples -- AIS

WRITTEN BY PATRICK BAUDELAIRE, JAY ISRAEL AND ROBERT SPROULL

FEBRUARY 1977

REVISED BY KEITH KNOX

MAY 1980

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

This document is for Xerox internal use only

ABSTRACT

This document describes a file format for the digital encoding of images, a basic Alto software package for manipulating these files, and a general purpose facility for user interaction and automatic replay.

KEY WORDS AND PHRASES

AIS, Array of Intensity Samples, picture processing, Alto, DIALOG package, DIRECTOR subsystem.

CR CATEGORIES

3.9, 4.4

Table of Contents

1. Overview

"Array of Intensity Samples" (AIS) is a standard format for the digital representation of images. By using this standard, researchers involved in diverse projects can more readily exchange images and software for processing them. The software described here for manipulating AIS files is a "starter set" that we hope will grow as various research groups write routines to add to it. We invite additions to the software repertoire, and shall endeavor to disseminate new software and to update this document accordingly.

1.1 AIS Files

"AIS" refers both to a standard digital format for image representation and to a collection of programs which manipulate files in that format.

Each AIS file is a digital representation of a rectangular array of *pixels* ("picture elements"). The array is a sequence of *scan-lines*. Each scan-line is a sequence of *pixels*. Each pixel is described by one or more *samples* -- more than one sample is required whenever several signals (e.g., chromatic separations) are to be represented in the same file.

The AIS format accommodates images of different sizes, different resolutions, different intensity quantizations, different numbers of signals, etc. A header on the file describes the various parameters of the file, called the *attributes*. The AIS format can also accommodate different methods of encoding images, but only one such encoding has been defined and implemented to date: an uncompressed array of signal samples (UCA encoding). Section **5** contains detailed specifications of the AIS format.

There are two classes of software described in this document.

- Programs and subroutines for manipulating AIS files.
- General-purpose user interface programs: DIALOG package and DIRECTOR subsystem.

All the software assumes a standard Alto configuration and one or two optional Trident disks. AIS files may be stored and manipulated on both types of Alto disks, Diablo and Trident. The Trident disk is required for printing AIS files via the Slot interface.

1.2 AIS Software

The software for manipulating AIS files falls into two classes:

- A package of AIS subroutines for carrying out individual actions: they are described in Section **3**.
- A driver subsystem AIS.run which interacts with the user to specify and perform a set of standard operations on AIS files. The driver subsystem incorporates both the AIS subroutines and the DIALOG package. It permits a user to manipulate AIS files without doing any additional programming. Details can be found in Section **2**.

The AIS package has two important properties that even the non-programming user must know. First, AIS subroutines may access not only disk-resident AIS files, but also *pseudo-files* which permit to use the Alto display (as if writing on an AIS file), or certain program-generated test patterns (as if reading from an AIS file). Second, AIS files are accessed via *windows*, imaginary rectangular apertures superimposed over the picture array, which facilitate processing a portion of an AIS picture. Figure 1 illustrates this abstraction and the coordinate systems used (one for the file as a whole and one for each window). A window may cover an entire file.

1.3 DIALOG Package and DIRECTOR Subsystem

The DIALOG package is a general-purpose set of run-time subroutines which reside between application routines and the user. DIALOG serves the following purposes:

- Provide a simple, common software utility for the user dialog, as it applies to the specification of input values for a subsystem, using the Alto keyboard and textual display.

- Permit input to come from a *command file* (also called a *script*), with gaps (if any) in the command file being satisfied by prompting the human user.

- Produce an *output log*, which includes the input command file information (if any), updated according to the values assigned during that session. Log file and command file have the same format. Thus the log can be used as a command file on a later occasion, with input values it contains being used automatically: the user will not be asked for them again. This is illustrated in Figure 2 -- the broken line indicates the log file being used as the command file during a subsequent session.

- Permit a *dry run* mode, in which the subsystem carries out the user dialog, but does not perform processing. The output log from such a run can be used as a command input file on a later occasion, causing the user-supplied values to be recapitulated selectively without human intervention. This, in fact, is how the initial command file can be created -- a separate file building/editing step is unnecessary.

- Provide a way to accommodate future procedures which take input from devices other than the keyboard.

The purpose of the DIRECTOR subsystem is to provide a mechanism for running a series of application subsystems in a controlled way. DIALOG and DIRECTOR cooperate to maintain the flow of control from one subsystem to another. DIRECTOR invokes each subsystem in turn, passing it a command file on which to operate. Upon completion of the subsystem, DIALOG transfers control back to DIRECTOR to determine the next step. It is important to note that the DIALOG subroutine package is the only part of the application subsystem which has any interface with DIRECTOR, and the only part which is aware of the source of the input parameters (disk file or interactive).

Section **4** describes the DIALOG subroutines and the DIRECTOR subsystem in more detail.

1.4 About reading this document

Some material is duplicated in two different sections so that portions of the document can be read independently. The non-programming user of the AIS subsystem may get started by reading section **2** (The User's View) and glancing over the description of the AIS format in section **5**. A more expert use of the subsystem (but still without programming) will require some knowledge about the

AIS subroutines, described in section **3**. The AIS programmer should get well acquainted with section **3**, and of course section **5**. Reading of section **4** (DIALOG and DIRECTOR) is required only for the programmer who wants to use the DIALOG package.

Certain notational conventions should be kept in mind when reading this document, especially the procedure calling sequences in Sections **3** and **4**. All software terms (file names, subroutine and subsystem names, calling sequences, parameter names, reserved names) are in a sans-serif font. Names of parameters are lower case, except that when a name is made up of several words the first letter of each word after the first is capitalized: scanDirection. A further exception is that words which are acronyms are capitalized: AIS, UCA. Names of procedures follow a similar convention except that the first letter is *always* capitalized: CreateAISDisplayFile. The calling sequences are in BCPL format. "_" is used to indicate the result of a function which returns a value. Square brackets indicate default values. An argument followed by [exp] defaults to exp if omitted or set to zero. An argument followed by [...exp] defaults to exp if omitted.

1.5 Related Documents

"Alto: A Personal Computer System, Hardware Manual" describes the standard hardware assumed by the AIS software.

"Alto Operating System Reference Manual," describes the software environment in which AIS programs reside.

"BCPL Reference Manual", by J. Curry, et al, describes the programming language used in implementing AIS and DIALOG software.

"TFS," by J. Melvin, found in "Alto Software Subsystems," describes the software package which the AIS routines rely on to access Trident disks.

"PREPRESS documentation," memo from R. Sproull, includes instructions for creating and modifying the .CU format font files used by the halftone procedure.

"Slot/3100 PRESS Printer Operation Procedures," memo from R. Sproull, describes how to print files which are in PRESS format.

"BRAVO Manual," by B. W. Lampson, describes a text editor suitable for editing DIALOG command files. Reprinted in the Alto User's Handbook.

1.6 Acknowledgements

B. Parsley programmed the RotateAIS routine described below, and offered numerous suggestions.

2. The User's View

This section explains how to use the driver subsystem AIS.run. There are two ways to run it. The first way is to type "AIS" to the Alto EXECUTIVE: this will permit one operation to be performed. The second way is to type "DIRECTOR" to the Alto EXECUTIVE; this permits a chain of operations to be performed, one step at a time. More about this later (**2.3**). Let us first describe the individual operations that AIS.run can execute (**2.2**).

2.1 Keying Conventions

Since both AIS.run and DIRECTOR.run use the DIALOG package, let us start by describing the user input conventions.

Each time the program needs information, it displays a message saying what it wants, and a cursor appears in the form of a black rectangle. This is a signal for you to type a response. Since a response may occupy more than one line, it is *always* terminated by ESC (not RETURN). Depending on the nature of the information, the response could be:

- One integer or more integers, separated by space, comma, or RETURN. They will normally be interpreted as decimal; to use octal notation, type "#" as a prefix or "B" as a suffix (e.g., 25 = #31 = 31B).
- A text string: any sequence of characters.
- A file name, input as a text string. However AIS.run applies certain defaulting conventions to AIS file names, which are discussed later.
- A yes or no answer: type "Y ESC" or "N ESC".
- A multiple choice selection: indicate your choice by typing one or more of the beginning characters of one of the alternatives, as needed to identifying it uniquely(see the example on page 11).
- A request for help: type "? ESC". Some information will be displayed for you, and you will be prompted again.

Certain keys have special actions:

ESC	terminates each response (not RETURN)
BS	backspaces one character
DEL	backspaces to the beginning of the response
CTRL-Q	aborts execution of the subsystem

Responses may often have *default* values. They are shown between curly brakets: {.....}. To specify that the default value be used, type ESC alone. Defaults have been selected to be values that you would want to use most of the time (hopefully). Sometimes, defaults are not provided. If you are unsure about what to type or what the default is, type "? ESC" to inquire.

2.2 AIS Subsystem

The purpose of the subsystem AIS.run is to permit a user to utilize the AIS subroutines without having to do additional programming. The discussion in this subsection is intended to give you information about the operations provided and how to use them. The seven basic operations are:

Copy	Merges *windows* of AIS files.
Legend	Places identifying text on an AIS *window*.
Reformat	Puts certain non-AIS *files* into AIS format.
Print	Prints an AIS *file* via the Slot interface.

Each time the AIS subsystem is activated, one operation is performed. You are prompted by a line listing the available operations: select one operation by typing one or more characters to identify it uniquely, terminated by ESC. The first listed operation is usually the default: it may be invoked by simply typing ESC. For instance, to invoke the following actions, it is sufficient to type:

Copy	ESC
Attributes	AT ESC
Print	P ESC

Some operations may in turn offer a succession of choices. Each new choice is selected in a similar fashion.

Before describing the details of each operations, let us first discuss some general conventions.

Some Conventions and Terminology

Files and windows

Two operations deal with AIS *files*: Reformat, Print. The remaining two operations deal with *windows* on AIS files: Copy (in all of its variants) and Legend. As has been mentioned earlier, windows are rectangular apertures superimposed over the picture array, which permit processing of selected portions of an AIS picture.

All the details of the AIS terminology are covered in section 5. Figure 1 also illustrates the concept of window, and shows the numbering scheme used for pixels and scan-lines. Here let us only mention that there may be up to four samples per pixel (numbered 0, 1, 2, 3) and from one to 16 bits per sample. Let us also define the term *scan direction*; this is a number that indicates how the scan-lines and pixels of the raster relate to the page image itself. With the Alto display, for example, the most natural scan direction is 3 -- the standard television raster. For the Slot/3100 printer, the scan direction is 8 -- pixels go from bottom to top; scan-lines from left to right. See Figure 3 for an illustration of these and other permissible scan directions.

File names

Every file (including AIS files) stored by an Alto is identified by a name, consisting of one or more parts, separated by dots. The last part, called the *extension* is customarily used to designate the format of the file. Example: Picture.AIS. No distinction is made between upper and lower case letters; thus the name picTure.ais refers to the same file as Picture.AIS. AIS software also permits a file name prefix indicating the device on which an AIS file is stored. The prefixes are S0:, S1:, T0:, and T1:, representing, respectively, the first system (i.e., Diablo) disk, the second system disk, the first Trident disk, and the second Trident disk. For example, S0:Picture.AIS and T0:Picture.AIS represent different physical files. When the prefix of an AIS file is omitted, the AIS software assumes a default disk. The default disk is T0:, if the Trident disk is installed and ready; otherwise it is S0:. Prefix S: is equivalent to S0:, and T: is equivalent to T0:.

Currently, disk names are used in a slightly different manner from that described above. The

second system disk is not implemented at all in any AIS software. The terms T0 and T1 do not refer to the actual disk drives, but rather to the two Trident drives or file systems initialized in the call to InitAIS, see section 3.2. For AIS.RUN, only one Trident drive can be initialized at a time. The drive is specified by a global switch giving the drive number. For example, if the command line were AIS.RUN/403, then the default disk and T0 would both refer to the second file system of a T300 on drive number 3.

It is not necessary to type the full name of an AIS file if it has the standard extension ".AIS". Suppose that you type the name "PICTURE", and assume that you are using the Trident disk. The subsystem will first attempt to find the file T0:PICTURE. (with a null extension). If it fails, it will then look for the file T0:PICTURE.AIS, and only then report failure. When creating a new file, the extension ".AIS" will automatically be appended to any file name without extension. If you explicitly want a file name *without extension*, you should type a dot at the end of the name: "PICTURE.".

Patterns

Some operations (Copy, Print) can be applied to *AIS patterns*, as well as regular AIS files. AIS patterns are software generated AIS pictures which can be used as read-only AIS files. Patterns are invoked by typing ESC rather than a file name. Then you will have to specify raster parameters as for a window on a regular AIS file: the only difference is that the height of the window is defined by a *count* of scan-lines rather than by a first and last scan-line; similarly, the window width is defined by a pixel count. Four types of pattern are implemented:

Constant	The same pixel repeated throughout the pattern.
Grid	Equally spaced horizontal and vertical lines. You must specify the spacing and the line thickness in each of the two directions.
Rectangles	A regular array of rectangles, each differing in intensity from its neighbor by an increment you specify. You will also be prompted for the rectangle dimensions.
Wedge	A rectangular array with intensity varying gradually along the scan-line between two extremes you specify.

Description of the Operations

Copy

This is the most extensive operation. Each copying step takes a source window, manipulates it in some way, and stores it in a destination window. The source window may be from a disk-resident file or it may be a pattern. The destination window may similarly be either disk-resident or on the Alto display.

If you want to act on the entire source file (or pattern), answer yes (i.e Y ESC) to the question "*Should the window be the whole AIS picture*?"; otherwise, you will be asked to specify a window inside the source AIS picture.

Similarly, when creating a *new* file as the destination of a Copy operation, the new AIS picture can be made equal to or larger than the destination window; in the later case, the new AIS picture will

be set to zero outside the destination window. In this case, you also have the option of changing the *padding* and *blocking* parameters: to understand what this means, consult the details of the AIS format (section **5**) and the AIS subroutines (section **3**).

Copy does a straightforward pixel-by-pixel combination of source and destination windows. The two windows should be of the same size.

There are eight options for treating the old values of the destination window (note: the bit-by-bit options are useful primarily for one-bit-per-sample AIS pictures).

Opaque	Overwrite the old values.
Paint	Overwrite only where a source value is non-zero.
Mask	Where source value is all ones, use destination value; elsewhere, zero (bit-by-bit logical "and").
Blend	Perform bit-by-bit logical "or."
Add	Perform the arithmetic sum.
Subtract	Perform the arithmetic difference.
Invert	Where source value is all ones, change destination zeros to ones and vice versa; where source value is zero, leave destination value unchanged (bit-by-bit logical "xor").
Compare	Compare source and destination *bit-by-bit*; result is ones where matches occur and zeros elsewhere.

A typical application would be to replace a window on an AIS picture by its negative (in the photometric sense): use the window as the destination of a Copy with an appropriately sized window of a constant pattern with sample value(s) of -1; if the file has one bit per sample, use the Invert option; otherwise, the Subtract option.

The source sample values may be *mapped* via table lookup before they affect the destination file. There are three ways to specify the mapping table: a one-for-one value substitution, dividing source values into value ranges and specifying a mapped value for each range, or specifying that equal ranges be used for dividing input values among mapped values (approximately linear mapping). When you have specified a map, you will be given the option of saving it in a file so that you will not have to describe it again if you want to use the same map in a subsequent session.

Reformat

The purpose of this operation is to help you convert between AIS and other image formats. There are three types of conversion supported.

AIS to PRESS

The source must be a window on an AIS file having one sample per pixel and either one or eight bits per sample. The output Press file may be B&W or color and may contain a PressEdit arrow. Either the image data itself or a pointer to a file containing the image data is written into the Press file.

PRESS printer output to AIS format

This method of conversion from PRESS to AIS format requires that the PRESS printer subsystem be run previously. One page of an intermediate bit map left behind in the file "Press.Bits" from the most recently printed PRESS file is converted to AIS format. The Press.Bits file may be in either SLOT or ORBIT format.

Other format to AIS

The source image file must include raster data already packed in UCA-type format. In this case, Reformat simply constructs the control information in the form of an AIS header and copies the raster data, skipping over any header present in the source file. To use this operation, you will need to be familiar with the layout of the source file.

Legend

This operation imprints text on an AIS window using crude characters. The intent is to place identifying labels on test images, not to provide pleasing textual appearance.

Print

Causes one or several AIS files to be printed via the Slot interface. Depending on the printer, you may print 1, 2, 4, or 8 bit-per-pixel images. The number of bits per pixel in the image must match the printer specification. No halftoning will be done at print time. You will be asked to specify the number of copies desired. There is a limited amount of freedom in printing AIS files. Each scan-line on the page may be printed once, or it may be doubled. A leading margin of scan-lines may be specified. Each scan-line on the printed page is divided into a number of pixels which you specify. This number includes (a) the image data, (b) a leading margin which you specify, and (c) a trailing margin (whatever is left). In allocating these margins, account for a few pixels which the output scanner traverses beyond the edges of the paper before encountering start & end of scan detectors. The AIS subsystem uses default values for these parameters which are found in your Alto user profile user.cm shown below. There should be 7 entries in user.cm. For instance, the defaults for the standard 3100 configuration are:

```
[AIS]
Double: 0
BitsPerPixel: 1
ScanMargin: 36
PixelMargin: 14
ScanLength: 4272
ScansPerPage: 3264
PixelsPerPage: 4224
```

Some of the necessary printing parameters can be specified only in the user.cm file, some only at run time and some can be defined both ways.

The parameters ScanMargin and PixelMargin, can only be set within the user.cm file. They are used to center the laser page image on the paper. The last two parameters, ScansPerPage and PixelsPerPage, are also specified in the user.cm file. They define the maximum size of the page image. The actual picture is located within this page image.

At run time, you are asked how many scan-lines and pixels to skip within the page image before printing the picture. The default values for these parameters are calculated by:

scan-lines in leading margin = (ScansPerPage - PictureHeight)/2
pixels in leading margin = (PixelsPerPage - PictureWidth)/2

The rest of the above parameters can be specified either in the user.cm file or at run time. If Double is non-zero, the scan-lines will be doubled. The number of bits per pixels that the printer can print is given by BitsPerPixel. ScanLength defines the total number of pixels across a scan-line and therefore defines the horizontal resolution.

Under a number of circumstances, the Print operation may reformat the AIS files, creating temporary files T0:print.ais, T0:prin1.ais, ... , T0:prin9.ais. In particular, printing must be done from a *contiguous* file on the Trident disk. This is the most common reason for reformating. If you want to save the files you have just printed so that it will not need reformating next time, copy the corresponding temporary files T0:prin*.ais using the command copy/C of the subsystem TFU. Here is an example of interaction with the Print command (prompting and messages by the AIS subsystem is in *small sans-serif italics*, user input is in **bold face**):

>**AIS RETURN**
AIS version 3.2 in control
Copy? Legend? Reformat? Print? {Copy}
p ESC
Print
Number of files to print {1}
ESC
Source file name (for pattern, just hit <ESC>) {}
picture ESC
Using file name: picture.AIS
How many copies? {1}
5 ESC
Bits/pixel for this printer {1}
ESC
Hardcopy scan-line length in pixels {4272}
ESC
Should each scan-line be doubled? {no}
ESC
Scan-lines in leading margin {1504}
ESC
Pixels in leading margin {1984}
ESC
Your file is being reformated for printing. If you want to save it, copy T0:print.ais

>**TFU COPY/C printPicture.ais _ print.ais RETURN**

The user requested 5 copies of the file picture.ais to be printed. The reformated file was copied under the new name printPicture.ais. Default values were used.

2.3 Using a Script

The AIS subsystem may be run with input taken from a command file (or script). As mentioned before, this permits automatic replay of a running session with identical, or perhaps-- at your discretion-- with slightly different, input. It may also be run under control of the DIRECTOR, to perform a specific series of operations prepared in a script.

The key is therefore the preparation of a script. Fortunately, this is made easy by the fact that the AIS subsystem (as any DIALOG-based system) produces an output log which records all the interaction that takes place during execution. This log is produced, whether or not the subsystem is run under DIRECTOR control. Output log and script have identical format: they are text files that are easily read and modified with a text editor. Scripts are usually prepared by modifying an existing script or an output log.

An Example of a Log File

The following log file was generated by a previous version of the AIS subsystem. Please note that several of the options presented there, are no longer available in AIS.RUN.

The format of the log file and script is formally described in section **4.3**. Here, it will be sufficient to give an example. The following is the log file generated by the AIS subsystem for the session given as an example above (it is usually called Dialog.out). For clarity of reading, line numbers have been added, and various fonts have been used to distinguish KEY WORDS, *messages and input prompting*, and **user input**.

```
1:    COMMAND SUBSYSTEM "AIS" VERSION "2.0" MODE live-run #
2:    PROMPT STRING "Copy? Delete? Attributes? Annotate? Legend? Reformat?
      Print? Show?" VALUE att #
3:    PROMPT STRING "File name" VALUE PICTURE #
4:    TERMINATION HOW successful #

5:    COMMAND SUBSYSTEM "AIS" VERSION "2.0" MODE live-run #
6:    PROMPT STRING "Copy? Delete? Attributes? Annotate? Legend? Reformat?
      Print? Show?" VALUE #
7:    PROMPT STRING "Source file name (for pattern, just hit <ESC>)."
      VALUE PICTURE #
8:    PROMPT STRING "Should the window be the whole image?" VALUE #
9:    PROMPT STRING "Merge? Halftone? Rotate? Zoom?" VALUE Z #
10:   PROMPT STRING "Destination file name" VALUE TEMPFILE #
11:   PROMPT STRING "Count of scan-lines" VALUE 512 #
12:   PROMPT STRING "Pixels per scan-line" VALUE 512 #
13:   PROMPT STRING "Raster direction" VALUE #
14:   PROMPT STRING "Samples per pixel" VALUE #
15:   PROMPT STRING "Words per scan-line" VALUE #
16:   PROMPT STRING "For blocking, give scan-lines per block. (0= no
      blocking.)" VALUE #
17:   PROMPT STRING "Should the window be the whole AIS picture?" VALUE #
18:   TERMINATION HOW successful #

19:   COMMAND SUBSYSTEM "AIS" VERSION "2.0" MODE live-run #
20:   PROMPT STRING "Copy? Delete? Attributes? Annotate? Legend? Reformat?
```

```
        Print? Show?" VALUE s #
21:     PROMPT STRING "Source file name (for pattern, just hit <ESC>)."
        VALUE TEMPFILE #
22:     PROMPT STRING "Should the window be the whole AIS picture?" VALUE #
23:     TERMINATION HOW successful #

24:     COMMAND SUBSYSTEM "AIS" VERSION "2.0" MODE live-run #
25:     PROMPT STRING "Copy? Delete? Attributes? Annotate? Legend? Reformat?
        Print? Show?" VALUE D #
26:     PROMPT STRING "File name" VALUE TEMPFILE #
27:     TERMINATION HOW successful #
```

The reader will easy recognize the correspondance between the log file, the display interaction presented above, and the four AIS operations that it represents:

Step	Items	Operation
1	1-4	Display attributes of file **Picture.ais**
2	5-18	Copy/Zoom **Picture.ais** to new file **TempFile.ais**.
3	19-23	Display file **TempFile.ais** on the Alto screen.
4	24-27	Delete file **TempFile.ais**

A few words about the content of this ouptut log. Each recorded unit of interaction is called an *item*. Each item ends with the character #. The first item of each step (items 1, 5, 19, and 24) shows the name and version of the program that wrote the script step, and indicates that it was a live run. The last item of each step (items 4, 18, 23, and 27) indicates that execution terminated successfully. Each of the other items shows (after the keyword STRING) a prompt displayed to the user and (after the keyword VALUE) the user's response. Note that in items 6, 8, 13, 14, 15, 16 17, and 22, there is no user response. This means that the user hit the ESC key alone to specify that the *default* value be used. When the file shown is used as a command file, the default will be taken automatically for those steps.

Let us now describe how to use such a log for preparing a script, and how to use a script.

Replaying a Program

Each segment of the output log delimited thus:

```
COMMAND .... #
  : : : : :
TERMINATION  ....  #
```

represents the running of a program. Extract such a segment and save it in the file Dialog.in. Activate the program again and it will run automatically without human intervention. Figure 2 illustrates what you have just done. The broken line shows a log file being used subsequently to substitute for keying, thus enabling unattended operation.

There is an important consequence: if you want to run the AIS subsystem entirely under your control, make sure to delete the file Dialog.in.

Making It Happen Differently Next Time

Using a text editor, a command file segment may be modified to cause the program to run differently next time. The usual modifications are of the following type:

- To cause a value to be requested from the user at a keyboard: delete the keyword VALUE and everything between it and the following #. (The # itself must remain, preceded by at least one space.)

- To cause a default to be used: the same as above, but leave the keyword VALUE.

- To give the user extra information: insert before VALUE (or before the closing # in the COMMAND item) something of the form
  REMARK "*This is additional information*"

- To use a different file as a log instead of Dialog.out (for instance AISlog.out), insert
  LOG AISlog.out
  before the closing # of the COMMAND item. Caution: the log file and command file must be different.

- To change the mode: in the COMMAND item, replace
  MODE *dry-run*
  by
  MODE *live-run*
  or vice versa. (In dry-run mode, the log file is written, but the operation specified is not performed.)

Suppose a user wanted to include steps 2, 3, and 4, of the log example above in a script, so that the source file name and the dimensions of the destination file may vary from session to session. The user wants only those three values to be input from the keyboard in the future. Suppose further that a reminder to the user is desired, before he/she keys in the file name, about what is going to happen to that file. Simply extract items 5 to 27 from the log, and modify items 7, 11, and 12. Using a text editor, it is straightforward to change the log to look like this:

```
                :               :
                :               :
7:      PROMPT STRING "Source file name (for pattern, just hit <ESC>)."
        REMARK "The picture will be displayed on the screen."  #
                :               :
                :               :
11:     PROMPT STRING "Count of scan-lines" VALUE  #
12:     PROMPT STRING "Pixels per scan-line" VALUE  #
                :               :
                :               :
```

Scripts with Many Operations

If you want to use a script containing several operations or involving several subsystems, you will need to run under control of the DIRECTOR subsystem. The script is prepared by pasting together segments of other scripts or log files, and perhaps modifying them as described above.

To cause a script to be executed, type "DIRECTOR" to the Alto EXECUTIVE. DIRECTOR will first ask for the file name of your script. (Typing conventions for responding to DIRECTOR are the same as those described above.) You will then be given an opportunity to override the modes (dry-run and live-run) contained in the script. Next, you may specify whether or not you want DIRECTOR to pause for you to intervene after each step (i.e., each invocation of the AIS subsystem). Execution now begins. Interaction with DIRECTOR would look like (prompting and messages by the DIRECTOR subsystem is in *small sans-serif italics*, user input is in **bold face**):

>**DIRECTOR RETURN**
DIRECTOR version 1.2 in control
If you want to use an existing script, name it.
Otherwise, just hit <ESC>. {}
ais.script ESC
To override script's modes, type L or D for live or dry run
Otherwise, just hit <ESC>. {}
ESC
If you want a special log file, name it {Dialog.out}
ESC
Do you want to pause after each step? {no}
ESC
>AIS
. . . . . .

If a subsystem aborts for some reason, or if you specified a pause after each operation, DIRECTOR provides the following options:

- go on to the next segment,
- repeat the segment with all input values provided interactively (values in the script being ignored),
- repeat it in the same mode as the original attempt,
- quit altogether.

You will recognize an abort by noticing the screen go blank for an instant, followed by the appearance of "SWAT" at the top of the screen and a message at the bottom of the screen explaining the situation. Type CTRL-K when you have read the message.

To help you get started, a skeletal script is provided in file AIS.script. It includes a sequence of activations of the AIS subsystem (described below), with no specific operations or values pre-assigned. When you want to carry out some AIS operations but do not have a relevant script available, simply activate DIRECTOR and specify AIS.script. The log file is cumulative; it grows every time a DIALOG-based subsystem is run. Delete it every so often so as not to consume too much disk space.

2.4 Getting started

In order to run the AIS subsystem, you will need the following files on your Alto disk:

AIS.run	the subsystem itself,
AIS.errors	a file containing messages that are displayed when the subsystem runs into error conditions.

If you intend to use the Print command, your file user.cm should contains the appropriate entries. The file AIS-usercm.slice contains a template of entries for printing on the Slot 3100 in its standard configuration. Simply insert AIS-usercm.slice into your file user.cm with a text editor. For other

Slot printer configurations, edit the entries as appropriate.

If you want to use DIRECTOR, get also the following files:

DIRECTOR.run	the DIRECTOR subsystem,
Dialog.errors	the error message for the DIALOG package,
AIS.script	a minimal script for running the AIS subsystem with all input from the keyboard.

All the files mentioned above are kept on the <AIS> directory on the central PARC computer MAXC. If you are connected to MAXC via FTP, you can get the latest version of the AIS subsystem on your disk by obtaining the file <AIS>AIS.cm, and then typing "@AIS.cm@" to the Alto EXECUTIVE. The same action is appropriate whenever a new version of AIS is released.

3. AIS Subroutines

This section describes a set of utility subroutines to deal with AIS files.

3.1 Conventions and Terminology

Each AIS file is a digital representation of a rectangular array of *pixels* ("picture elements"). The array is a sequence of *scan-lines*; each scan-line is a sequence of pixels. Each pixel is described by one or more *samples* -- more than one sample is required whenever several signals (e.g., chromatic separations) are to be represented.

AIS accommodates images of different sizes, different resolutions, different intensity quantizations, different numbers of signals, etc. A *header* on the file (the attribute section) describes the various parameters of the file. AIS can also accommodate different methods of encoding images, but only one such encoding has been defined to date: an uncompressed array of signal samples.

Not only may disk-resident AIS files be accessed, but pseudo-files are defined for accessing the Alto display or certain program-generated test patterns. Each AIS file represents a rectangular image. Files (and pseudo-files) are accessed via windows, imaginary rectangular apertures which facilitate processing a portion of an image. Figure 1 illustrates this abstraction and the coordinate systems used (one for the file as a whole and one for each window). A window may cover an entire file.

Reading and writing

Reading and writing of AIS files can be performed at three different levels: a sample at a time, a scan-line at a time, or several scan-lines at a time. The first is the slowest, but conserves space and is suitable when the processing involved does not proceed in raster fashion. The second is the most commonly used. The third is usually significantly faster than the second when the window is as wide (in the pixel direction) as the entire file. When reading a scan-line at a time, there are two modes available: packed and unpacked. When unpacked mode is used, the calling routine's data image has sample values separated -- one sample per word. This representation is independent of the encoding method used in the file itself. When packed mode is used, the calling routine's data image is in the same form as the data on the disk. Programs using this mode require less buffer space, but may not be applicable to new coding types. In addition to the three basic data access levels, there are facilities for operating on an entire window. Included here are routines for merging, rotating, printing, displaying, zooming, and halftoning images. These are currently a somewhat limited set of facilities, intended to grow with usage.

File name

Every file (including AIS files) stored by an Alto is identified by a name, consisting of one or more parts, separated by dots ("."). The last part, called the *extension* is customarily used to designate the format of the file. Example: Picture.AIS. No distinction is made between upper and lower case letters; thus the name picTure.ais refers to the same file as Picture.AIS. AIS software also permits a file name prefix indicating the device on which an AIS file is stored. The prefixes are S0:, S1:, T0:, and T1:, representing, respectively, the first system (i.e., Diablo) disk, the second system disk, the first Trident disk, and the second Trident disk. For example, S0:Picture.AIS and T0:Picture.AIS represent different physical files. When the prefix of an AIS file is omitted, the AIS software assumes a default disk. Normally, the default disk is T0:, if the Trident disk is installed and ready; otherwise it is S0:. Prefix S: is equivalent to S0:, and T: is equivalent to T0:.

Currently, disk names are used in a slightly different manner from that described above. The second system disk is not implemented at all in any AIS software. The terms T0 and T1 do not refer to the actual disk drives, but rather to the two Trident drives or file systems initialized in the call to InitAIS, see section 3.2. For AIS.RUN, only one Trident drive can be initialized at a time. The drive is specified by a global switch giving the drive number. For example, if the command line were AIS.RUN/403, then the default disk and T0 would both refer to the second file system of a T300 on drive number 3.

Errors

Whenever the AIS routines encounter a non-recoverable error (e.g., a hard equipment failure or a data inconsistency), the routine AISError is called. This routine halts the processing step and displays a message indicating the nature of the problem. The user types CTRL-P or CTRL-K to terminate the processing step cleanly. AISError is also available for use by programmers utilizing the AIS routines.

All the AIS.RUN software is written in BCPL.

3.2 Level 0 -- Basic AIS Routines.

zone_InitAIS(stackSpace [3000], driveNumberT0, driveNumberT1)

This procedure is called to initialize level 0 routines, and to create a zone from which *all* free storage will be seized (up to 32K words). The routine also initializes the Trident disk software and microcode. If the last two parameters are absent, only one Trident drive, TP0, will be initialized. If the arguments are present, they are interpreted as the drive numbers of the Tridents to be initialized as T0 and T1, respectively. For example, the call to InitAIS(3000, #403) will initialize TP403, i.e. the second file system of a T300 on drive number 3, as the default disk T0.

SetAISDefaultDisk(diskName)

Changes the default disk prefix for OpenAISFile and DeleteAISFile calls. Example: SetAISDefaultDisk("S0:").

AISError(code, arg1, arg2, ...)

This routine is called to signal an internal error (i.e., not an error in the user's responses, but a problem in equipment, data structure, or format). The file AIS.Errors contains a list of error messages (indexed by code) which will be displayed by SWAT.

Files

f_OpenAISFile(fileName, how [readOnly], rasterVec [0], attributeLength [1024])

This function opens an AIS file of the given name.
how determines the mode of access to the file. It has one of the reserved values: readOnly, readWrite, writeOnly or readWriteNew (meaning create a new file).
If the file is being created, rasterVec must be initialized with all the relevant parameters so that the file size can be determined, and so that it can be allocated on the disk. This rasterVec can be created by obtaining the parameters from an already existing file (see ExtractAISUCARaster, below), or by building it anew (see MakeAISUCARaster and UserAISUCARaster, below). attributeLength specifies how many words are to be allocated for the attribute section -- it will rounded upward (if necessary) to a multiple of 1024 words.
If the file already exists, rasterVec (unless it is zero or omitted) will be filled with the raster

part of the attribute section; attributeLength is ignored in this case.
The function returns the value f, which is to be used for all subsequent reference to the file; it returns 0 if it is unsuccessful. The notion of an AIS file is similar to the notion of a stream in the Alto operating system.

success_DeleteAISFile(fileName)
Deletes the AIS file of the given name. Returns true if a file was found, false otherwise. Notice: this is the unique exception where a file name is used. All other procedures use the value f returned by OpenAISFile.

length_ReadAISAttributes(f, partType, partVec [0])
If partVec is present and non-zero, this procedure treats it as a vector and reads into it the corresponding attribute part. partType may have one of the following 5 reserved values: placementPart, rasterPart, photometryPart, commentPart or allParts. If you wish to read or write your own private part type, be sure not to use any of the part types already defined in AISfile.d.
In any event, the value returned, length, is the number of words in the attribute part, or 0 if unsuccessful (i.e. the file does not contain the specified part type).

WriteAISAttributes(f, partType, partVec)
Writes the corresponding attribute part into the file referred to by f.
Exception: the raster part may be written only when creating a file using OpenAISFile(fileName, readWriteNew, rasterVec).

CloseAISFile(f)
Closes the file f, and all windows associated with it (see below).

s_GetAISStream(f)
This function returns the disk stream that is open for accessing the file referred to by f. Manipulate the stream at your own risk.

Windows

w_OpenAISWindow(f, firstScan [...0], lastScan [...scanCount-1], firstPixel [...0], lastPixel [...scanLength-1], unPack [...false], whichSamples [...allOnes])
This call creates a window on the AIS file specified by f, and in effect establishes a local coordinate system for dealing with the AIS. In referencing the window, the first scan-line will be numbered 0; the last, lastScan-firstScan. Similarly, the first and last pixel are numbered 0 and lastPixel-firstPixel, respectively. This function also sets the mode in which scan-lines will be read and/or written.
unpack specifies whether scan-lines are to be read as they are represented on the disk, or unpacked one sample per word. On writing, similarly, unpack=true means that the samples in the user's data vector are interpreted as unpacked values: one sample per word.
whichSamples is a bit mask that tells which samples in the pixel should be returned by ReadAISScanLine or written by WriteAISScanLine: if bit 0 (i.e., the leftmost or most significant bit) is on, the first sample will be read; if bit 1 is on, the second, etc. Restriction: if unpack is false, *all* samples must be selected.
The default window, obtained by calling this procedure with only one argument f, covers the whole AIS file.

w_OpenAISWindowFromVec(parameterVec)
Same as above, with parameters in a vector.

At any given time, a file is permitted to have several windows. When determining the size of a data vector required for reading and writing scan-lines through a window, you should obtain the value vLengthMin using GetAISWindowParams or SetAISWindowParams. For accessing a single scan-line, the smallest data vector allowed is vLengthMin words. For multiple scan-lines, multiply this by the number of scan-lines. AIS software selects vLengthMin according to certain static characteristics of the window and file.

CloseAISWindow(w)
Releases a window.

vLengthMin_GetAISWindowParams(w, parameterVec [0])
This function extracts the seven parameters of the window in the order given to OpenAISWindow, and returns them in parameterVec if this argument is provided: parameterVec!0 is f, parameterVec!1 is firstScan, etc.

vLengthMin_SetAISWindowParams(w, unpack [false], whichSamples [allOnes])
This function changes the indicated window parameters.

Reading and writing

Several procedures are provided for reading and writing single or multiple window scan-lines, as well as individual samples. With a window is associated the notion of a *current* scan-line, which is the scan-line of the most recent read or write activity. The scan-line number argument sl provided to reading and writing procedures, may have the special value -1, meaning *next* scan-line (i.e., the current scan-line + 1). Immediately after opening a window, next scan-line will be the first scan-line in the window (i.e., sl=0). The read and write routines do no scaling of the sample values (i.e., they ignore the information in the photometry part).

boolean_EndofAISWindow(w)
Returns true if the current scan-line is the last scan-line in the window (i.e. lastScan-firstScan).

ReadAISScanLine(w, sl, vLength, v0, v1, v2, ...)
WriteAISScanLine(w, sl, vLength, v0, v1, v2, ...)
These functions transfer scan-lines between user's vectors and the window, according to the mode set.
sl is the scan-line number in the windowed coordinate system.
vLength is the length (in words) of the vectors v0, v1,...
The calls specify as many vectors as needed; remember that the whichSamples bits specify which samples will be returned. If unpack is false, all samples must be selected; the scan-line is read or written in packed format, appropriately shifted, into or from vector v0. If unpack is true, there should be as many vectors as samples are requested: for instance, if whichSamples is equal to #060000, the second and third samples will be read into or written from vectors v0 and v1 respectively. However, if one of these vectors is zero, the corresponding sample will not be read or written.
These procedures generate an error if sl is out of range or if vLength is too small (i.e. vLength < vLengthMin).

numberRead_ReadAISScanLineS(w, slBegin, count, vLength, v)
numberWritten_WriteAISScanLineS(w, slBegin, count, vLength, v)

These procedures can be used to transfer several *packed* scan-lines to and from the window. These calls are usually substantially faster than loops using ReadAISScanLine or WriteAISScanLine in those cases when the window is as wide as the full AIS file (i.e., firstPixel=0 and lastPixel=scanCount-1). Otherwise, speed is the same as that of iterated calls to ReadAISScanLine or WriteAISScanLine.

The procedures return the number of scan-lines successfully read or written. However, they will generate an error if the initial scan-line, slBegin, is out of range, or if vLength is too small (i.e. vLength < count * vLengthMin).

value_ReadAISSample(w, sl, pixel, sample [0])
WriteAISSample(w, value, sl, pixel, sample [0])

Functions for reading and writing individual samples, which work somewhat more slowly than the scan-line versions above. The sample number, which defaults to specify the first sample, is provided so that if there are several samples per pixel, you can in fact access each in turn. These procedures will generate an error if scan-line number, pixel number, or sample number is out of range.

Cursor

The AIS read/write routines incorporate a feature which illustrates on the Alto screen the progress of a series of operations. The feature is activated by setting the external static AISCursor to a non-zero value. If AISCursor is non-zero, every call to a read or write procedure sets the Alto screen cursor to look like a window roller-shade showing proportionally the current scan-line position in the AIS window being accessed. The shade is rolled up when the window is just opened, the current scan-line being zero; the shade is pulled down when the current scan-line is at the end of the window. If AISCursor is neither zero nor -1, it should point to a 16-word cursor bit map, which will replace the cursor after each call to CloseAISWindow.

An implementation detail

AIS files and windows are "objects" for which several of the level 0 AIS routines are generic routines; pointers to them are planted in the object itself. The routines are:

File routines:

CloseAISFile	Call0
OpenAISWindow	Call1
ReadAISAttributes	Call2
WriteAISAttributes	Call3
GetAISStream	Call4

Window routines:

CloseAISWindow	Call0
GetAISWindowParams	Call1
SetAISWindowParams	Call2
EndofAISWindow	Call3
ReadAISScanLine	Call4
WriteAISScanLine	Call5
ReadAISScanLineS	Call6
WriteAISScanLineS	Call7
ReadAISSample	Call8
WriteAISSample	Call9

Miscellaneous routines

m_Minimum(a, b) = (a < b) ? a, b

M_Maximum(a, b) = (a > b) ? a, b

DoubleMultiply(r, a, b)
puts the double precision product of a and b in the two-word vector r.

q_DoubleDivide(r,a)
returns r/a where r is a two-word vector and a is a single-word value.

r_MulDiv(a,b,c)
returns (a*b)/c, the intermediate product being computed in double precision.

q_IntDivide(a, b) = ia/bj = (a+b-1)/b
computes the quotient of a and b rounded upward, assuming a and b positive.

3.3 Level 1 -- Type Dependent Routines.

AIS files of type UCA

UCA coding type is described in Section **5**. Generic procedures for this type are implemented for samplesperPixel <_ 4.

MakeAISUCARaster(rasterVec, scanCount, scanLength, scanDirection [3], samplesperPixel [1], bitsperSample [1], wordsperSL [minWordsperSL], scanLinesperBlock [allOnes], paddingperBlock [allOnes])

This routine puts the appropriate raster part of type UCA in rasterVec. The minimum value for wordsperSL is computed by:
minWordsperSL=IntDivide(scanLength*samplesperPixel, 16/bitsperSample)

ExtractAISUCARaster(w, rasterVec)

This routine extracts the shape of the window w, and its mode, and creates a raster part that can be used to create an AIS file that will hold w. This function is useful when extracting a piece of an AIS. The raster parameters wordsperSL, scanLinesperBlock and paddingperBlock are not copied from w; rather, they are given the default values shown in the preceding paragraph.

AIS pseudo-files

There are several kinds of pseudo-files. Each of these can be used the same way f is used in the calls above, with a few exceptions specified below.

f_CreateAISDisplayFile(displayLines [as much as storage permits], width [606], scanDirection [3], reverse [false], zone [the zone created by InitAIS])

This function creates an "AIS file" on the Alto display. Programs can write and read scan-lines from windows on it (1 bit per point) and have the image appear appropriately. The displayLines parameter states how many Alto display scan-lines are to be allocated. The scanDirection parameter governs how the scan-lines will be laid down in the display itself; it must be either 3 or 8. If reverse is false, a sample of value 1 is displayed as a white dot, and the picture background is black, and vice versa.
Exception: GetAISStream returns the address of the display bitmap (manipulate it at your

own risk); ReadAISAttributes deals only with the raster part; WriteAISAttributes is a no-op.

f_CreateAISPattern(patternType, rasterVec, arg0, arg1, arg2, ...)

This is a way of making a read-only AIS for returning various test patterns (e.g., constant gray, a grid, etc.). rasterVec is of type UCA. Windows may be opened on patterns; this gives the programmer an opportunity to set window size and to specify the reading mode. In the discussion below, the term sampleVec is used to denote an array of samples; it is employed because a pixel may have several samples in it. (For one sample per pixel, sampleVec!0 is the sample; hence the routines can be called with lv value as a sampleVec argument.) The actual number of samples returned is determined in the same way as when reading from disk-resident files. The patterns presently available are:

patternConstant	arg0 is a sampleVec for a constant pixel to be spread throughout the window.
patternGrid	arg0 is the number of pixels per grid unit arg1 is the number of scan-lines per grid unit arg2 is the number of pixels in a grid line (i.e., its thickness) arg3 is the number of scan-lines in a grid line arg4 is the sampleVec for pixels off grid lines [allOnes] arg5 is the sampleVec for pixels on grid lines [0].
patternRectangles	arg0 is the number of pixels per rectangle arg1 is the number of scan-lines per rectangle arg2 is the sampleVec for the first rectangle [0] arg3 is an incremental sampleVec to step by with each new rectangle [allOnes].
patternWedge	arg0 is the sampleVec at the first pixel [0] arg1 is the sampleVec at the last pixel [allOnes].

Exceptions: in accessing patterns, WriteAISAttributes is a no-op; GetAISStream yields a "stream" which will cause an error if ever used; and ReadAISAttributes deals only with the raster part.

3.4 Level 2 -- User Conveniences.

This level includes the routines listed below. Each one whose name starts with "User" exploits the DIALOG package (see Section IV) to prompt the user for relevant parameters for constructing the object returned.

PrintAISPart(stream, partType, partVec)

Puts a textual representation of an attribute part on stream.

LegendAIS(w, slCount, string, sampleValue [allOnes])

This procedure imprints in crude characters, the given legend string on the AIS window w, using slCount scan-lines in which to do it. This is not intended to create pretty characters, but only to place identifying labels on test images. Uses the system font

scanDirection_UserAISScanDirection(promptString)

This function displays the promptString and solicits user input of a scan direction value. Only valid AIS scan directions are accepted.

UserAISUCARaster(rasterVec)

fills in the entries in rasterVec for a UCA raster by prompting the user to supply values to use. rasterVec may then be used to create a pattern or a disk AIS file.

maps_UserAISMap(window, zone)

The purpose of this procedure is to construct a map such as used as an optional argument in MergeAIS or HalfToneAIS (see below). First, however, it ascertains whether or not the user wants mapping of input values at all; if not, zero is returned. A map may be specified in one of three ways: a one-for-one value substitution, dividing the input values into ranges and giving a value for each range, or specifying that equal ranges be used for dividing input values among mapped values. window is the AIS window to which the map will be applied. zone is a main memory allocation zone from which the storage for maps will be obtained. The caller is responsible for freeing maps when it is no longer needed. Finally, the user is given the option of saving the map in a file, to be retrieved at a subsequent call to UserAISMap. The map file format is as follows:

word 0	A password (#61273) serving as a hint that this is a map file.
word 1	Size of the file in words (excluding words 0 and 1).
word 2-5	Offsets from word 2 for start of map for samples 0-3, respectively (zero when the corresponding sample is not mapped).
word 6-end	Lookup tables mapping input values to output values.

f_UserAISPattern()

is analogous to CreateAISPattern. The values required to specify the characteristics of the pattern are obtained through interaction with the user.

w_UserAISWindow(how, lv dispUse, lv dispWidth, lv dispHeight, lv dispRasDir, lv dispReverse)

This procedure interacts with the user to determine a file to open and a window to open on that file. how must be readOnly or writeOnly; in the latter case a new file will be created if the user names a file not already existing. In the readOnly case, a window on a pattern may be specified by the user. With writeOnly, the user may specify a display, but this is an exceptional case: the display file will not be created. Instead, the disp*xxx* variables are set. Each disp*xxx* is a pointer to a variable relevant to display usage. The procedure sets dispUse=true if the user specified display output; false otherwise. The remaining disp*xxx* variables will be set by this procedure whenever it sets dispUse. Each corresponds to the argument of CreateAISDisplayFile which its name resembles.

The rationale for making an exception of displays is as follows. Creating a display file involves allocating a bit map which requires large amounts of main memory. With UserAISWindow implemented as described here, the region occupied by the "User" and DIALOG routines can be freed to participate in the bit map. In this way, larger displays can be accommodated. The price is a slight inconvenience for the programmer calling UserAISWindow, who must subsequently test dispUse and, if it is true, call CreateAISDisplayFile and open a window on the result.

3.5 Level 3 -- Utilities.

This level contains some utility subroutines for operating on files or windows.

MergeAIS(wOut, wIn, operator [opStore], maps [0], keepLowerBits [false])

This procedure merges the input window wIn into the output window wOut. wIn and wOut may be of different sizes: however, the procedure will actually operate on two equal size sub-windows determined by the minimum width and height of wIn and wOut.
The whichSamples parameters of the two windows will determine the correspondance between input samples and output samples. If the sample sizes of wOut is smaller than that of wIn, the parameter keepLowerBits comes into play: if it is true, the low order bits of each sample of wIn are used (the excess high order bits being discarded in copying); if it is false, only the high order bits are used.
The value of operator determines how old values in wOut are to be treated:

opStore: overwrite old values.
opNew: wherever a new value is non-zero, overwrite.
opAdd, opSub: add or subtract old values.
opAnd, opOr, opXor, opEqv: bit-wise logical operations (useful primarily for one-bit-per-sample AIS pictures).

If the maps argument is non-zero, the sample values from wIn are mapped into new values, as follows:
outVal=(if maps!sample eq 0 then inVal else (maps!sample)!inVal).
This can be used for modifying the photometric scale through table lookup (for instance, tone reproduction correction).

AISPress(wIn, streamOut, PRESSFileName, resolution)

This procedure converts from AIS to PRESS format.
wIn must be a window of an AIS file having one or eight bits per sample, and one sample per pixel.
streamOut is a stream already open on the disk file in which the PRESS output is desired.
PRESSFileName and resolution are used for PRESS control information: PRESSFileName is a BCPL string identifying the file; resolution is in pixel per inch.

AISGetPressPage(AISFileName, pageNumber [1])

This routine provides a mechanism for obtaining an AIS file from an image that starts as a PRESS file page. The PRESS Printer subsystem must be run before calling this subroutine. AISGetPressPage locates the file Press.bits (intermediate data left as a by-product of the PRESS Printer), extracts page pageNumber, converts that page to AIS format, and stores the result in AISFileName.

PrintAIS(fvec, copies [1], printerBits [1], slLength [4416], slDouble [false], bitsInLeadingMargin [20], scanLinesInLeadingMargin [20], nFiles [0])

This procedure prints one or several AIS files using the Alto Slot interface. The files must have one sample per pixel and may have 1, 2, 4, or 8 bits per pixel.
If nFiles=0, fvec is a single AIS file; otherwise, fvec is a vector of length nFiles containing the AIS files.
There is a limited amount of freedom in printing AIS files. Each scan-line on the page may be printed once, or it may be doubled, governed by the boolean slDouble. A leading margin of scan-lines may be specified (scanLinesInLeadingMargin). Each scan-line on the printed page is divided into a number of pixels which you specify (slLength). This number includes (a) the image data, (b) a leading margin (pixelsInLeadingMargin), and (c) a trailing

margin (whatever is left). In allocating these margins, account for a few pixels which the output scanner traverses beyond the edges of the paper.
Under any of the following circumstances, PrintAIS reformats the files (creating the intermediate file T0:print.ais, T0:prin1.ais, ..., T0:prin9.ais) to facilitate printing:

- the file is a pattern.
- the file is not on a Trident disk.
- the file is not contiguous.
- the number of words per scan-line is odd.
- the larger of block size and file size is greater than 8,192 words.
- the photometry part is missing or 0 for 1 bit per pixel files.
- the photometry sense is 1 for >1 bit per pixel files.

Warning: use of this routine requires knowledge of the code. It is expected that printing requirements will be satisfied by the Print command of the AIS subsystem (section **2.2**).

3.6 Modules and Files

The AIS subroutines are found in the following files, all stored on the <AIS> directory on PARC central computer MAXC.

Minimum basic Level 0 routines for AIS disk files*:*

AISUCA0	InitAIS, SetAISDefaultDisk, OpenAISFile, DeleteAISFile, OpenAISWindowFromVec, IntDivide, DoubleMultiply, DoubleDivide, MulDiv, Minimum, Maximum, AISError, some generic routines for disk windows.
AISUCA1	generic routines for disk file data reading and writing.
AISUCA2	MakeAISUCARaster, ExtractAISUCARaster, generic routines for manipulating attributes.

Required when using *display* pseudo-files:

AISDisp	CreateAISDisplayFile, generic routines for display pseudo-files.

Required when using *pattern* pseudo-files:

AISPat	CreateAISPattern, generic routines for pattern pseudo-files.

Required for special processing steps:

AISPrompt1	UserAISWindow, UserAISUCARaster.
AISPrompt2	UserAISPattern, UserAISMap, UserAISScanDirection.
AISUser	PrintAISPart, LegendAIS.
AISMerge	MergeAIS
AISPrint	PrintAIS
AISGetPress	AISGetPressPage
AISPress	AISPress
AISSlotProc	a subroutine of PrintAIS borrowed from the PRESS Printer subsystem.

Additional routines for implementing the AIS subsystem described in Section **2** are found in the following files:

AIS	main program.
AISChatter	conducts user dialog.
AISPChatter	conducts user dialog for the printing command.
AISReformat	puts certain non-AIS files into AIS format.
Template	contains a general-purpose text stream output formatting subroutine.

Definition files necessary for compiling the source files listed above:

AISFile.d	structure and manifest declarations used by all AIS software.
SlotDefs.d	structure and manifest declarations relevant to AISPrint.

Files needed for *running* AIS software:

AIS.errors	error messages displayed through SWAT.
AISSlotMc .br	microcode for running both the Slot/3100 and the Trident disks.

Useful *command* files:

AISbldr.cm	loader command for combining relocatable files to form the AIS subsystem.
AISInstall.cm	a command file using FTP to retrieve all the necessary relocatable modules and subsystems from MAXC. The relocatable modules are stored together in the file <AIS>AIS.dm.
AIS.cm	a command file for retrieving the AIS and DIRECTOR subsystems.

Other useful files:

AIS.script	a skeletal script which DIRECTOR can use to sequence through a number of AIS steps.
AIS.updates	a text file listing the changes to the AIS software.

4. DIALOG and DIRECTOR

The DIALOG package is a general-purpose set of run-time subroutines which reside between application routines and the user. DIALOG serves the following purposes:

- Provide simple, common utilities for the user dialogue, as it applies to the specification of input values for a subsystem, using the Alto keyboard and textual display.

- Permit input to come from a *command file* (also called a *script*), with any gaps in the command file being satisfied by prompting the human user.

- Produce an *output log*, which includes input command file information (if any), updated according to the values assigned during that session. This log can be used as an input command file on a later occasion, with input values it contains being used automatically: the user will not be asked for them again. This is illustrated in Figure 2.

- Permit a *dry run* mode, in which the subsystem carries out the user dialog, but does not perform processing. The output log from such a run can be used as a command file on a later occasion, causing the user-supplied values to be recapitulated without human intervention. This, in fact, is how the initial command file can be created -- a separate file building/editing step is unnecessary.

- Provide a mechanism for the later attachment of procedures which take input from devices other than the keyboard.

The purpose of the DIRECTOR subsystem is to provide a mechanism for running a series of application subsystems in a controlled way. DIALOG and DIRECTOR cooperate to maintain the flow of control from one subsystem to another. DIRECTOR invokes each subsystem in turn, passing it an input command file on which to operate. The subsystem may take some input values from the command file and some from a human operator. DIALOG creates an output log in a format suitable for use as a command file in a later session. Upon completion of the subsystem, DIALOG transfers control back to DIRECTOR to determine the next step. It is important to note that the DIALOG subroutine package is the only part of the application subsystem which has any interface with DIRECTOR, and the only part which is aware of the source of the input parameters (disk file or interactive).

Subsystems operating under the control of the DIRECTOR need not use the DIALOG subroutines, but only by using DIALOG are the advantages offered by DIRECTOR exploited. The compatibility facility is provided primarily for convenience so that existing subsystems can be intermixed with DIALOG subsystems and be run under the control of DIRECTOR. Conversely, DIALOG may operate without DIRECTOR. DIALOG and DIRECTOR are written in BCPL.

4.1 Overview of DIALOG

This section discusses the general sequence of events in the execution of a subsystem which utilizes the DIALOG package. Each of the two subroutines -- DialogInit and Prompt -- are described in more detail in a later section. The DIALOG software and command file formats permit a separate subsystem (DIRECTOR) to run automatically several programs or subsystems in succession. The DIALOG package is most appropriate for subsystems which can be provided with all their required input values first, and which then go ahead and perform the required processing. It is less

appropriate for subsystems which require continual close user interaction (e.g., text editors).

The sequence of events in the execution of a subsystem is as follows:

1. The subsystem is invoked by any of the standard methods. It performs its own internal initializations.

2. The subsystem calls DialogInit, which causes the DIALOG software to be initialized.

3. For each input value required by the subsystem, the following series of steps occurs:

3.1. The subsystem calls the Prompt routine.

3.2. The Prompt routine looks in the *script* (if any) first for an appropriate value. If present, that value is returned to the subsystem without human intervention. If the value is absent, the user is prompted to supply it. The user's input is subjected to a cursory validity check (to catch blatant typos). The user also has the option of (a) specifying that a default value be used, or (b) causing the subsystem to be aborted.

3.3. The Prompt routine appends to the *output log* an item indicating the value used, or that the default was taken.

3.4. The Prompt routine returns the value to the calling routine. The calling routine is not informed where the value originated (file or human).

4. After all required values have been processed, the subsystem performs its main function. An exception is the dry run mode; upon detecting this mode, the subsystem is responsible for skipping this step altogether. (The DIALOG package itself does not cause the skip to happen, but does provide the mode information. In fact, the DIALOG routines do not take any different action in dry run mode from live run mode.)

5. Finally, the subsystem terminates execution in any of the standard ways (e.g., BCPL finish or abort)

If the output log is used as script on a subsequent run of the subsystem, the same values will be supplied, but without the human interaction. Moreover, this automatic supplying of values can be selective. If the log is edited to remove the value(s) associated with a particular prompt, the human interaction will occur for that prompt.

4.2 The User's View

This section describes what the user of a DIALOG-based subsystem sees. A later section deals with the programmer's interface.

Running a program

A subsystem is activated either through DIRECTOR (described in more detail in section **4.5**) or, more conventionally, by typing its name to the Alto EXECUTIVE.

Each time the program needs information, it displays a message saying what it wants, and a cursor appears in the form of a black rectangle. This is a signal for you to type a response. Since a

response may occupy more than one line, it is *always* terminated by ESC (not RETURN). Depending on the nature of the information, the response could be:

- One integer or more integers, separated by space, comma, or RETURN, terminated by ESC. They will normally be interpreted as decimal; to use octal notation, type "#" as a prefix or "B" as a suffix (e.g., 25 = #31 = 31B).
- A text string (for instance a file name): any sequence of characters, terminated by ESC.
- A yes or no answer (i.e. a boolean value): type "Y ESC" or "N ESC".
- A request for help: type "? ESC". Some information will be displayed for you, and you will be prompted again.

Certain keys have special actions:

ESC	terminates each response (not RETURN)
BS	backspaces one character
DEL	backspaces to the beginning of the response
CTRL-Q	aborts execution of the subsystem

Response may often have *default* values. They are shown between curly brakets: {.....}. To specify that the default value be used, type ESC alone. Sometimes, defaults are not provided. If you are unsure about what to type or what the default is, type "? ESC" to inquire.

Replaying a program

After running a program, look at the log file (usually Dialog.out) using a text editor. An example of such a log file is given in section **2.3**. Each segment delimited thus:

```
COMMAND .... #
. . . . .
TERMINATION  ....  #
```

represents the running of a program: all the user interaction with the program has been recorded there, and it is sufficient for running the program again, automatically, with the same input. Extract such a segment and save it in the file Dialog.in. Activate the subsystem again and it will run again, repeating the same actions, this time without human intervention: it is now using the file Dialog.in as a script. Figure 2 illustrates this -- the broken line represents taking a log file for use as a command file in a later session. Below, in the discussion of DIRECTOR, it is described how to chain together many such segments into a more complicated script.

Making it happen differently next time

Using a text editor, you may modify a script or a log file, in order to run a program again with different input. The following section explains the details of the command file and log file format. Here are some simple ways of changing a command file, causing a program to run in a slightly different fashion the subsequent time:

- To cause a value to be requested from the user at a keyboard: delete the keyword VALUE and everything between it and the following #. (The # itself must remain, preceded by at least one space.)

- To cause a default to be used: the same as above, but leave the keyword VALUE.

- To give the user extra information: insert before VALUE (or before the closing # in the COMMAND item) something of the form
  REMARK "This is some extra information"

- To use a different file as a log instead of Dialog.out (for instance MyOwnLog.out), insert:
  LOG MyOwnLog.out
  before the closing # of the COMMAND item.
  Caution: the log file and command file must be different.

- To change the mode: in the COMMAND item, replace
  MODE dry-run
  by
  MODE live-run
  or vice versa.

4.3 Format of Command File and Log File

The command file (or script) is a text file, and therefore it can be read easily by somebody interested in its contents, and can be edited using standard text editing programs. The file is divided into *items*. Each item consists of one or more complete lines. The first line of each item starts with a keyword specifying the item type. The last line ends with a space followed by the special character #. There are three item types. The first item of each file is the *command* item; the last, the *termination* item. Each intermediate item is of the third type, a *prompt* item. Each prompt item corresponds to one call to the Prompt procedure.

The syntax of the command file items is described below: keywords are denoted by words in SMALL CAPITALS, parameters are in *tiny italics*. The first parameter of each item is required; the others are optional: some are defaulted when omitted.

Conventions for text strings

When a character string appears in the command file, it is enclosed in double-quote characters, and "*" is an escape character as in BCPL strings. That is, *t is interpreted as the tab character; *n and *c, carriage return; *l, line feed; *", double-quote character; and **, as * itself. For the character after "*", upper and lower case have the same effect. Strings (but not other types of values, e.g., an integer) in the command file may spill over from one line to the next.

Command item

COMMAND SUBSYSTEM *subsysString* VERSION *versionString* MODE *mode* LOG *fileName* REMARK *remarkString* #

subsysString is the name of the subsystem to which the file pertains.

versionString is the particular version of the subsystem to which the file pertains

mode is either dry-run or live-run. Default is live-run.

fileName is the name of the file to which the output log information is to be appended. Default is Dialog.out.

remarkString is an arbitrary string of commentary. If present, it is displayed to the user when the DialogInit routine is called.

This is an example:

```
COMMAND SUBSYSTEM "AIS" VERSION "2.0" MODE live-run
LOG AISlog.out REMARK "This is the February release" #
```

Prompt item

PROMPT STRING *promptString* REMARK *remarkString* VALUE *value* #

promptString is the explanation of the desired value displayed to the user when human input is requested. This string originated at a previous run of the subsystem as a subsystem-supplied prompt.

remarkString is a commentary which is also displayed to the user at the time of the prompt. It pertains specifically to the context of the particular input command file, as contrasted with *promptString*, which is relevant to all invocations of the subsystem.

value (if present) is an assignment which has already been made to the subsystem parameter involved. value can take different forms, as explained below: one or more integers, a string, a yes/no answer, etc. If the keyword VALUE and the value specification are both absent, it means that assignment to the parameter should be made by human intervention. If the keyword VALUE is present but only blanks appear between it and the closing #, it means that the subsystem-provided default value should be used, without human intervention.

Termination item

TERMINATION HOW *how* #

how is either successful or user-aborted or program-aborted.

Log file

A log file is produced automatically by the DIALOG package. It has exactly the same format as a command file, with the following exception: the parameters REMARK and LOG are not generated automatically, therefore they will be present in the log file only if they are present in the command file.

4.4 Subroutine Calls

The two DIALOG procedures are DialogInit and Prompt.

mode_DialogInit(userParams, subsysName, versionString, zone, inputCommandFile ["Dialog.in"])

userParams is the second argument handed to every subsystem activated by the Alto EXECUTIVE. The subsystem must pass this value on to the DIALOG package as the first argument of DialogInit.

subsysName is the name of the subsystem being executed, expressed as a BCPL string. DialogInit checks this name against the corresponding parameter in the input command file (if any). In case of a mismatch, the input file will be disregarded and the user will be prompted for each value.

versionString is the particular version of subsysName being executed. DialogInit compares it with the version parameter in the input command file. In the event of a mismatch, the user is warned, and execution otherwise proceeds normally.
zone is a standard main memory allocation zone from which the DIALOG package may obtain temporary working space.
inputCommandFile is the disk file read by the DIALOG package. If the specified file does not exist, the user will be prompted for each input value.
mode is dryRun or liveRun. If the initialization procedure does not know the mode from the input command file, liveRun is assumed.
The COMMAND item of the log file is now created and appended to the existing file. (DIALOG software always appends to the log file; it does not delete or overwrite material already there.)

firstVal_Prompt(promptString, type [typeBoolean], valueVector[0], defaultVector[0], proced[0])

promptString is a message indicating to the user the nature of the information required. It is expressed as a BCPL string.

type indicates the form of the information, and is one of the following:

typeInteger
typeIntegerPair
typeIntegerVector
typeBoolean (a yes/no answer, represented by the constants true and false)
typeString (a character string)
typeName
typeMessage
typeError

typeName is the same as typeString except that Prompt strips off leading blanks and everything starting with the first delimiter (,;/" SPACE RETURN LF TAB) following the name.
typeMessage and typeError cause output to be displayed, but no values to be sought. Their operation is identical, except for the extra features of typeError noted below.

valueVector (if present) points to a vector where the prompt procedure will place the parameters it obtains.
defaultVector (if present) specifies values to be placed in valueVector if the user indicates that default values be used.
proced is intended for use when input from a device other than the keyboard is required. It is the address of a procedure to supply the values.
firstVal is set by the Prompt procedure to valueVector!0 (or if valueVector is not present, whatever would have been valueVector!0). Its purpose is to permit use of the vector arguments to be avoided for simple inputs.

Some discussion of the format of valueVector and defaultVector is in order. The maximum length for each is 128 words. For types typeIntegerVector, typeString, and typeName, the first member specifies in binary integer format the number of additional members. (For the other types, there is no explicit count.) Thus, for typeIntegerVector, valueVector!0=count, and valueVector!1 through valueVector!count contain data. Likewise, for typeString and typeName, valueVector!0 rshift 8 = count, valueVector!0 & #377 is the first character, and the last character is in valueVector!(count/2) -- the left byte if count is even, the right if count is odd. (This string format is the same as that of BCPL strings.)
Remember also that the (actual or potential) valueVector!0 content is what is returned as

firstVal. For typeString and typeName, this is a word with count in the left byte and the first character in the right byte.

Operation of the prompt procedure is as follows. First, the promptString is compared with the corresponding expected *prompt item* in the command input file. If the strings match, the value(s) is(are) taken from the command file. Otherwise, the promptString is displayed to the user (along with any REMARK from the command file) and input is requested. If proced is absent, the keyboard is read, terminated by an ESC character.

If proced is present, it is called by the statement:

stringResult_proced(pointer, result)

It must place, in the vector pointed to by result, either (a) a character string, such as would be typed in by a user, in the form of a BCPL string < _ 254 characters long, or (b) a binary value coded according to the type-dependent valueVector conventions described above. stringResult is a boolean returned to indicate which of these applies: true for (a), false for (b). When stringResult=true, the string will be processed exactly as a key input, and the result will be validated. Otherwise, Prompt will do neither of these.
pointer is a mechanism whereby proced may access the arguments of Prompt. Specifically, pointer!0 is promptString, pointer!1 is type, pointer!2 is valueVector, pointer!3 is defaultVector, and pointer!4 is proced.

Except when proced is used and stringResult=false, the Prompt routine validates that the input conforms with the type and does any necessary conversions. In the event validation fails, the prompt cycle is carried out with human interaction (even if the value came from the command file). If the user or the command file specifies that the default be taken, the values from defaultVector are used (but are not checked). Now, regardless of the source of the values, the Prompt routine stores them in valueVector (if present), in the format described above. The first word is saved as the Prompt procedure's result. Prompt then appends to the output log a prompt item including the prompt string and the values used. In addition, any REMARK from the input command file is copied. If the value was taken from the input command file, that file is advanced to the next item; if that item is another PROMPT, it will be used at the next Prompt subroutine call.

The purpose of typeError is to deal with the case in which the subsystem detects a value inconsistency. The Prompt procedure accommodates this situation by backspacing the output log by one item whenever called with typeError. Prompt will not permit more than one typeError call consecutively without at least one intervening value assignment call. One final case concerns the user's indicating that the subsystem should be aborted. In this case, control is not returned to the calling routine; instead, Prompt causes execution of the subsystem to terminate.

4.5 DIRECTOR Subsystem

DIRECTOR works from a command file. This script is a concatenation of several portions of the output log described above. Each portion corresponds to the execution of a subsystem. The output log serves another valuable purpose. It is an audit trail recording the sequence of subsystem steps and input values of a given experiment or operation. It can be consulted whenever it is necessary to reconstruct the history of the operation, such as in debugging equipment or application programs. It can also provide for a replay of an operation for closer scrutiny. In the other extreme, DIRECTOR's use of a script permits an experiment or operation to be canned, along with repeatedly occurring input parameters, so it can run with little intervention at the keyboard, or even unattended.

A run of DIRECTOR proceeds according to the following steps.

1. User activates the DIRECTOR subsystem.

2. DIRECTOR prompts the user to supply the name of a file from which the script is to be read. Alternatively, the user may specify that this is a scriptwriting run. In this event, the user specifies a list of subsystem names, from which DIRECTOR constructs a skeletal command file for the run. Upon completion of the run, the output log will contain a completely filled out command file.

3. The user may select one of three run modes:

   dry run
   live run
   take modes from the command file (default)

In the first two cases, the run mode selected applies to all subsystems, and overrides whatever may be in the command file. In a scriptwriting run, a user selecting the third option will have to provide a run mode for each subsystem.

4. The user may select one of two continuity modes:

   auto run (proceed through the script without interruption)
   pause (halt after each subsystem for human intervention)

The latter capability is provided especially for the purpose of replaying an aborted run.

5. For each subsystem segment, the following steps are carried out

   5.1. DIRECTOR activates the subsystem and supplies a command file for DIALOG.

   5.2. The subsystem performs its processing, producing a log and finally returning control to DIRECTOR.

   5.3. DIRECTOR checks for successful completion of the subsystem. In the event it aborted, the user is given the option to either (1) go on to the next segment, or (2) repeat the segment with all input values to be provided interactively, or (3) repeat in the same mode as the original attempt, or (4) quit altogether.

   5.4. In the event of successful completion, DIRECTOR also checks the continuity mode. If it is pause, the user is given the same four options as in step 5.3.

6. DIRECTOR reaches the end of the script and releases control.

4.6 Modules and Files

The DIALOG package consists of the following source program files:

DialogInit	contains the subroutine of the same name.
Prompt	likewise.
PromptSubs	contains subroutines of Prompt.

DialogUtil	utility routines for the other modules.
Dialog.dfs	contains structure and manifest declarations for DIALOG and DIRECTOR.

Also required are:

Template	a general-purpose output formatting procedure.
Dialog.errors	error messages displayed through SWAT.

After DialogInit is called, the storage it occupies may be released. Likewise, after all values have been specified, Prompt and PromptSubs may be released. However, *DialogUtil may never be released* because it contains the code to wrap up a subsystem and return control to the DIRECTOR.

DIRECTOR consists of one source file (named Director) and requires the DIALOG package.

Other files of interest are:

DirectorBLDR.cm	loader command for combining relocatable files to form the DIRECTOR subsystem.
Director.state	used by DIRECTOR and DIALOG to exchange information.
Director.scratchRem	temporary holding place for the file Rem.cm.
Director.scratchScript	temporary script used in scriptwriting runs.

5. AIS Format

This section contains the detailed specifications for files in AIS format.

5.1 Terminology

An encoded image is a representation of an "array of intensity samples" (AIS), or a *raster*. For the formats specified here, the raster is a sequence of *scan-lines*; each scan-line is a sequence of *pixels*. The signal value(s) for a pixel are given by one or more *samples*. By convention, the numbering of these objects begins at 0: the first scan-line is numbered 0; the last of n is numbered n-1.

We shall assume standard Alto terminology. A *word* is 16 bits. A *file*, for the purposes here, is a homogeneous sequence of data bytes (the particular way these data bytes are stored on the disk itself, the way disk directories are built, etc. are of no concern to this discussion).

When coordinates on a page are required, we shall use the following coordinate system: the (0,0) point is at the lower left corner of a (portrait) page; the x direction is to the right; the y direction is up the page. The unit of measurement is the *mica*, equivalent to 10 microns.

Because of their convenience, BCPL structure declarations are used to describe some of the formats. These declarations show how successive bits and words of the file are laid out; they are not necessarily intended for explicit use in your programs. For more rigorous definitions of data structures, see file AISfile.d.

5.2 The AIS Format

The purpose of the AIS format is to define a rectangular raster region, and optionally give the desired placement of the region on a page. In addition, the encoding of the raster information must be specified. There is also a need to record photometry information concerning the measurement of the intensity samples.

See figure 5 for a graphical description of the AIS file format. An AIS file consists of an *attribute section* followed by a *raster section.* The attribute section describes the format of the raster and various other information about the data; the raster section gives the samples themselves.

The attribute section is always a multiple of 1024 words long, and is itself composed of several *parts*: the *page placement part*, the *raster part*, the *photometry part*, and the *comment part.* Only the raster part is mandatory: it must be at the beginning of the attribute section. The other parts may be in arbitrary order. Any unused words in the attribute section are set to 0 (this permits expansion later on). The first word of the attribute section is a password, which serves as a hint that the file is in AIS format. The second word is the length of the attribute section.

5.3 Raster Part

All information in the raster part is mandatory. It has two components:

Scan: This specifies how the image has been sampled and raster-scanned.

scanCount: The number of scan-lines of information that are recorded in the raster data.

scanLength: The number of pixels that are recorded for each scan-line.

scanDirection: This quantity describes how the scan-lines and pixels of the raster are to relate to the page image itself. The value in scanDirection is:

pixel-direction-description * 4 + scan-line-direction-description.:

A direction-description is:

0 = toward the right on the page
1 = toward the left on the page
2 = toward the top of the page
3 = toward the bottom of the page

Example: If the scanDirection=3, the pixel direction is to the right, and the scan-line direction is to the bottom. This means scanning begins at the upper left corner of the image area: pixels are placed at successive positions to the right; scan-lines descend the page (this is like TV video, or the way the Alto records bit maps). The scanDirection for the Slot/3100 printer is 8: pixels run up the page; successive scan-lines move to the right on the page. See Figure 3 for an illustration of acceptable values for scanDirection.

samplesPerPixel: Each pixel may have several signals associated with it; this entry records the number of samples associated with a pixel. For example, if the image is a color picture recorded with R, G and B signals, this value would be 3. Different coding schemes may conceivably place different interpretations on this value.

Coding: The coding information specifies how the intensity samples are actually encoded in the raster section. We may need to add new entries to the coding information as new coding formats become available; for this reason, the raster part is prefaced with a length to permit future expansion.

codingType: This word gives the "type" of the encoding. There is one type currently defined:

UnCompressedArray: codingType=UCACodingType=1. The samples are recorded as a sequence of fixed-size bytes. The samples for the first scan-line are recorded first, then the second scan-line, etc. Within each scan-line, the first pixel is recorded first, then the second pixel, etc. The additional coding part entries for this type are:

UCABitsperSample: The sample byte size, i.e. the number of bits recorded for each sample. The byte size may have any value between 1 and 16. If 16 is not a multiple of the byte size, bytes do not straddle word boundaries: they are right justified, with a few unused bits on the left. For instance, if UCABitsperSample is 5, there are 3 bytes per word, and the leftmost bit is unused.

UCAWordsperSL: The samples for an individual scan-line occupy UCAWordsperSL words in the file; scan-lines therefore begin on a word boundary. Zero bits are used to pad out space remaining in the last word of the scan-line. Any words beyond the minimum required to represent the scan-line are filled with zeros.

Note that, if UCABitsperSample is equal to 1, 2, 4, 8, or greater than 8, then:

UCAWordsperSL*16 $\geq$ scanLength * samplesPerPixel * UCABitsperSample.

UCASLperBlock: For certain real-time devices, it is essential that there be an occasional break in the encoding to facilitate buffering. If no such breaks occur in the encoding, this entry is -1. If the encoding is blocked, this entry gives the number of scan-lines per block. For example, if there are 32 scan-lines per block, then the 1st,

33rd, 65th, etc. scan-lines all begin at the start of a block. The number of words in a block is usually a multiple of the disk sector size (see below).

UCApaddingperBlock: Not all disks have 1024 words in a sector. It therefore is necessary that blocking units of other sizes be allowed. This information is recorded by specifying the number of padding words left over in each block. If the file is not blocked, then this entry is -1. From the previous example, if there are 32 scan-lines per block, then there are (32*UCAWordsperSL+UCApaddingperBlock) words in each block.

5.4 Placement Part

The placement part contains the coordinates required to specify the size and shape of the image region on a page. If placement is not specified, all four entries are -1.

placeXLeft: The x coordinate on the page of the lower left corner of the image rectangle.

placeYBottom: The y coordinate of the lower left corner of the image rectangle.

placeXWidth: The width (in micas) of the image rectangle on the page.

placeYHeight: The height (in micas) of the image rectangle on the page.

Note that the placement information, together with the raster part, effectively specifies a resolution at which the image is to be interpreted. However, most AIS programs will ignore the placement part when performing calculations, and believe instead the specification of the array as given by the raster part. The only point of the placement part is to permit printing location information to be specified.

5.5 Photometry Part

The purpose of the photometry part is to identify the signal that has been sampled and some of the conditions under which it was sampled.

photometrySignal: This entry is intended to give some idea of what signal has been sampled and recorded in the file:

unspecified = -1
specified in comment part = -2
Black and White = 0
Red separation = 1
Blue separation = 2
Green separation = 3
Cyan separation = 4
Magenta separation = 5
Yellow separation = 6
x signal (CIE) = 7
y signal (CIE) = 8
... (more to be added)

R,G,B samples, 3 per pixel = 100
C,M,Y samples, 3 per pixel = 101
C,M,Y,B samples, 4 per pixel = 102
Y,x,y samples, 3 per pixel = 103

... (more to be added)

photometrySense: This entry gives the "sense" of the samples: if larger values of sample indicate greater transmission or reflection, then the sense is 0. If larger values of sample indicate greater density or absorption, then the sense is 1. (Note: This value can be derived from the "scale" parameters, below, but is given here for convenience.)

photometryScale (together with pointA, ...B and ...C): These specify the conversion between sample values (viewed as integers) and the actual physical values. The photometryScale value gives the sort of scale used:

Z Reflectance or Transmittance X 1000=1

Z Optical Density X 1000=2

Points A, B, and C are three points on the curve defining the relationship between the actual physical values and the digital values. The first 16 bit word is 1000 X the actual value, e.g. reflectance (0-1000). The second word is the digital sample value, e.g. (0-255). If not all three points are needed, unused values are -1.

photometrySpotType, photometrySpotWidth, photometrySpotLength: These three numbers give the type and dimensions of the spot that was assumed when the image was created, measured as follows: the width is the dimension along the scan-line; the length is orthogonal to the scan-line. The dimension is measured from 20% energy points on the spot. The units are 100 times the dimensions expressed in pixels or scan-lines; the entries are -1 if no width information is known. The currently defined spot types are: undefined, defined in comments, rectangular and circular, see AISfile.d.

photometrySampleMin, photometrySampleMax: The minimum and maximum values of the sample integers (-1 if unknown).

photometryHistogram: If a histogram of the samples is available, it follows this entry, and this entry gives the length of the table (presumably 256 for 8-bit samples; a length of 0 or -1 indicates no histogram is present). Each table entry is one word which contains an integer that is 32767 times the frequency of occurrence of the corresponding sample. Note that this is a histogram of the sample values themselves, before any scaling.

5.6 Comment Part

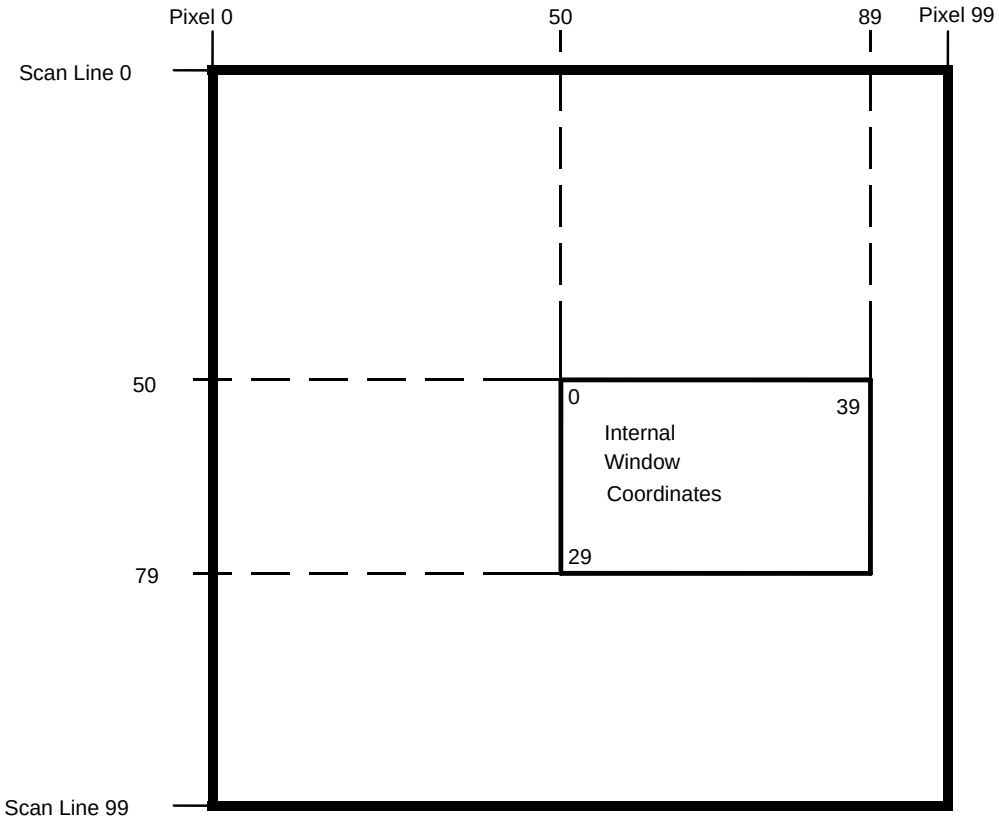
This part simply contains a text string in BCPL format. It is used to record an arbitrary comment pertaining to the AIS file such as: what type of picture it is, how and when it was generated, etc. The BCPL format limits the text string to 255 characters.

5.7 Declarations

The following declarations indicate the format of the attribute section. They are intended to be descriptive, not authoritative. The version used for programming is in file AISfile.d.

```
structure Attributes [
        password                                    word        // Password = -31574.
        attributeLength                             word        // Number of words before data begins
        body ^1,variable                            word        // Here are the "parts" (see below)
        remainder ^1,rest                           word        // Words of 0 to pad.
        ]

structure RasterPart [                                          // RASTER PART:
        @APH                                                    // Header: type=1; length=variable
        scanCount                                   word        // Number of scan-lines
        scanLength                                  word        // Pixels per scan-line
        scanDirection                               word        // Scanning directions
        samplesPerPixel                             word        // Number of samples
        codingType                                  word        // Method of coding (e.g., UCACodingType=1)
        other ^6,length                             word        // remainder of coding part (e.g., @UCA)
        ]

structure PlacementPart [                                       // PLACEMENT PART:
        @APH                                                    // Header: type=2; length=5
        placeXLeft                                  word        // Position of lower left corner
        placeYBottom                                word
        placeXWidth                                 word        // Size of image area on page
        placeYHeight                                word
        ]

structure PhotometryPart [                                      // PHOTOMETRY PART:
        @APH                                                    // Header: type=3; length=variable
        photometrySignal                            word        // Signal that is sampled
        photometrySense                             word        // Explained above.
        photometryScale                             word        // Scaling type
        photometryScaleA                            @VALUE      //    and arguments (points on
        photometryScaleB                            @VALUE      //       conversion curve)
        photometryScaleC                            @VALUE
        photometrySpotType                          word        // Spot type
        photometrySpotWidth                         word        // Spot size
        photometrySpotLength                        word        //      information
        photometrySampleMin                         word        // Sample range
        photometrySampleMax                         word        //      (if known)
        photometryHistogram                         word        // Number of words in histogram
        photometryHistData ^1,photometryHistogram   word
        ]

structure CommentPart [                                         // COMMENT PART:
        @APH                                                    // Header: type=4; length=variable
        @STRING                                                 // A text string in BCPL format
        ]

structure APH [                                                 // Attribute Part Header
        type                                        bit 6       // Type of attribute part.
        length                                      bit 10      // Length in words, including this one.
        ]

structure UCA [                                                 // UnCompressedArray
        UCABitsperSample                            word
        UCAWordsperSL                               word
        UCASLperBlock                               word
        UCApaddingperBlock                          word
        ]

structure STRING [                                              // BCPL string format
        length                                      byte
        char ^1,255                                 byte
        ]

structure VALUE [
        sample                                      word        // 1000*(actual sample value), e.g. reflectance (0-1000).
        level                                       word        // digital sample value, e.g. (0-255).
        ]

structure FL [                                                  // Proposed foating-point format:
```

```
sign                    bit         // Note: this format is identical to the high
exponent                bit 8       // order 32 bits of PDP-10 floating point.
mantissa                bit 23      // See DEC documentation.
]
```



This example shows a 100x100 AIS file containing a 40x30 window.

File parameters		Window parameters	
scanCount	100	firstScan	50
scanLength	100	lastScan	79
scanDirection	3	firstPixel	50
		lastPixel	89

Figure 1. An Illustration of an AIS Image

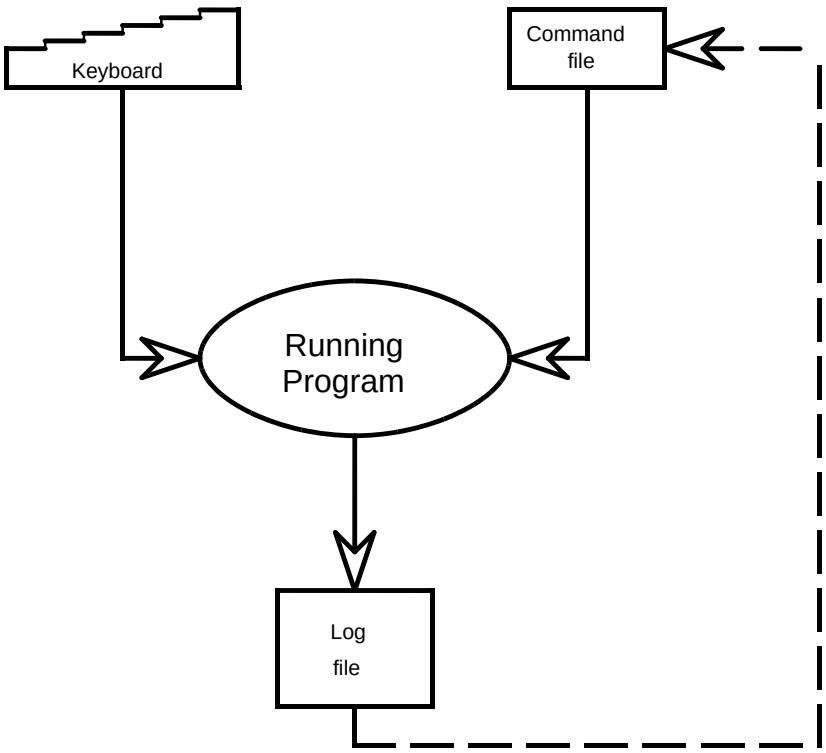


Figure 2. Usage of the Command and Log Files

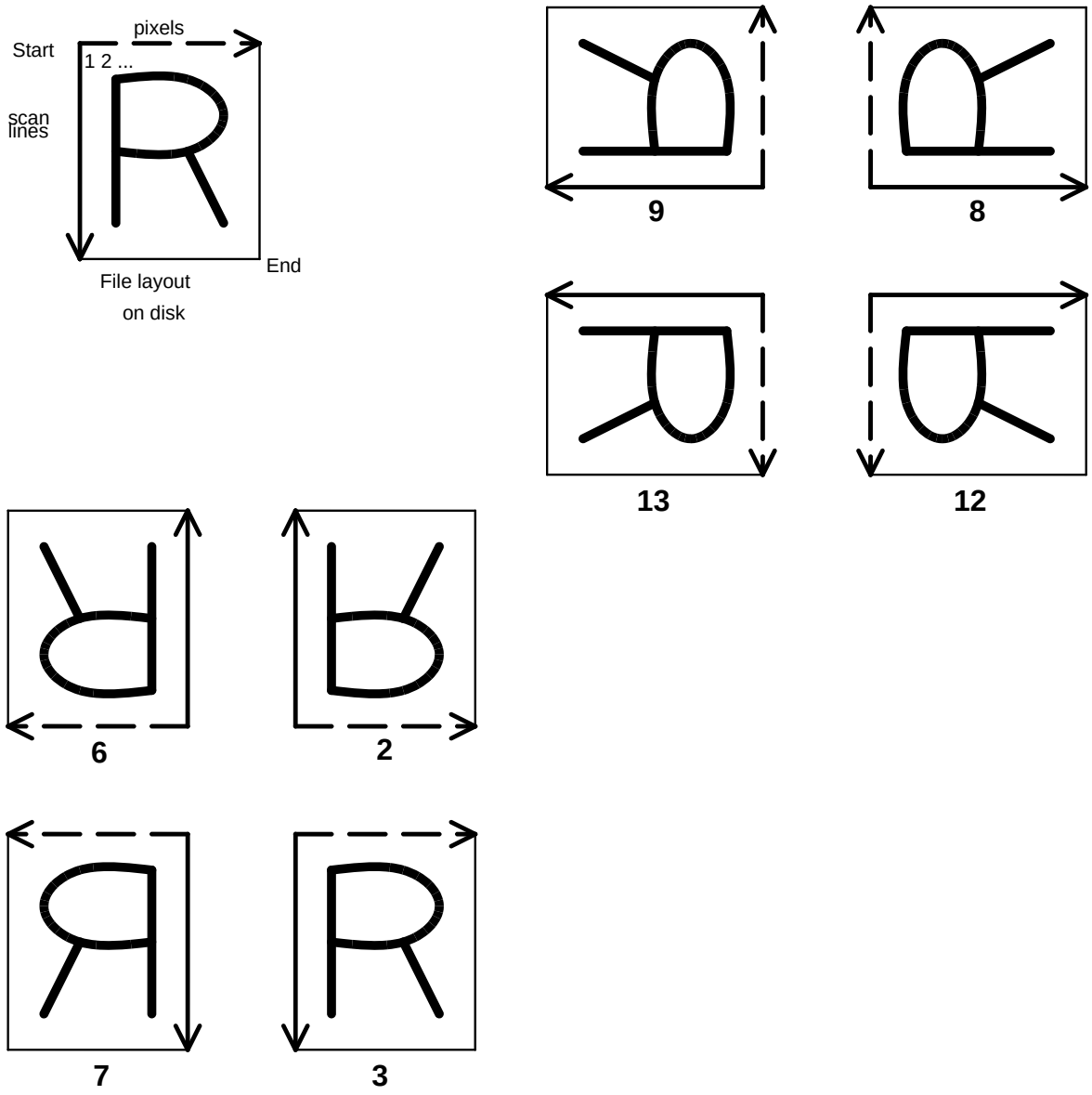


Figure 3. AIS Scan Directions

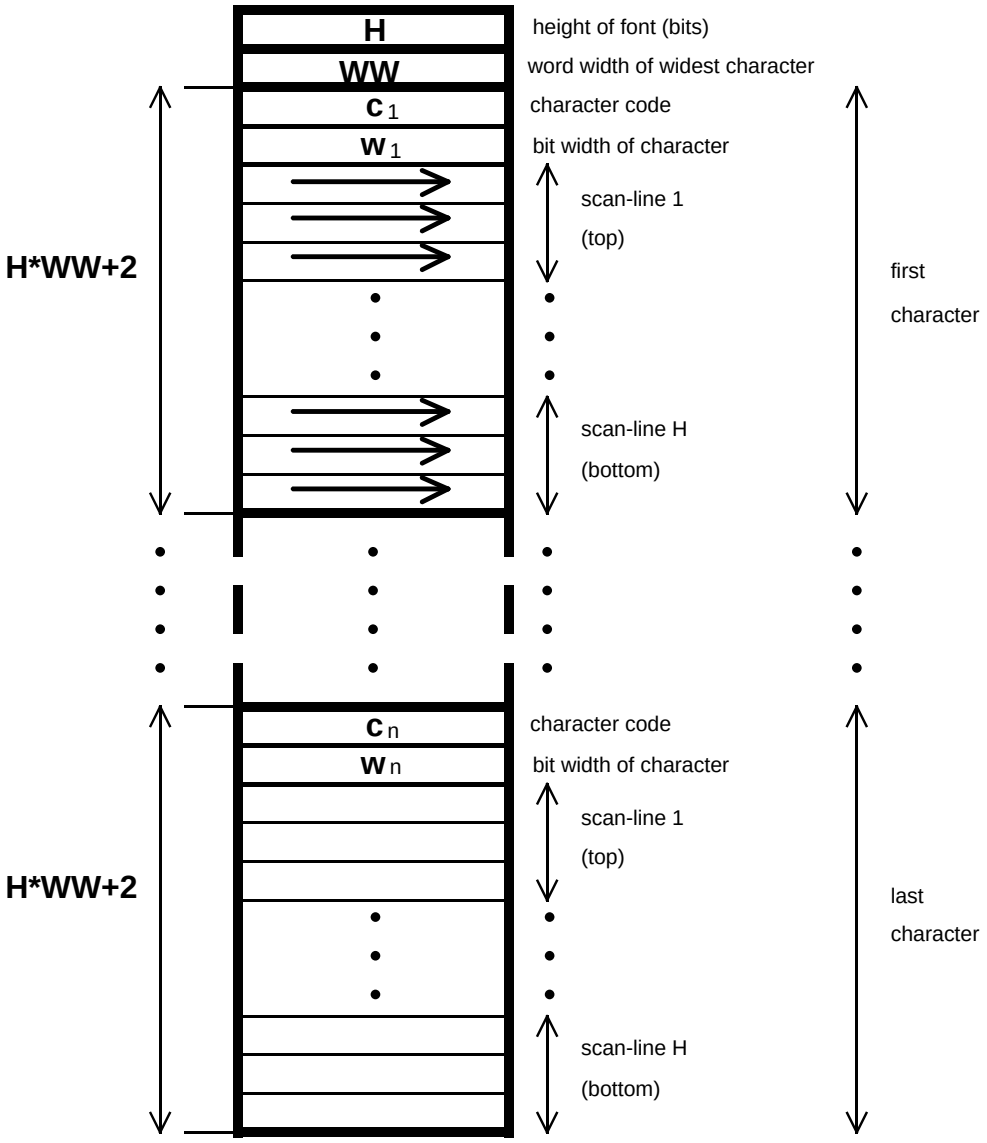
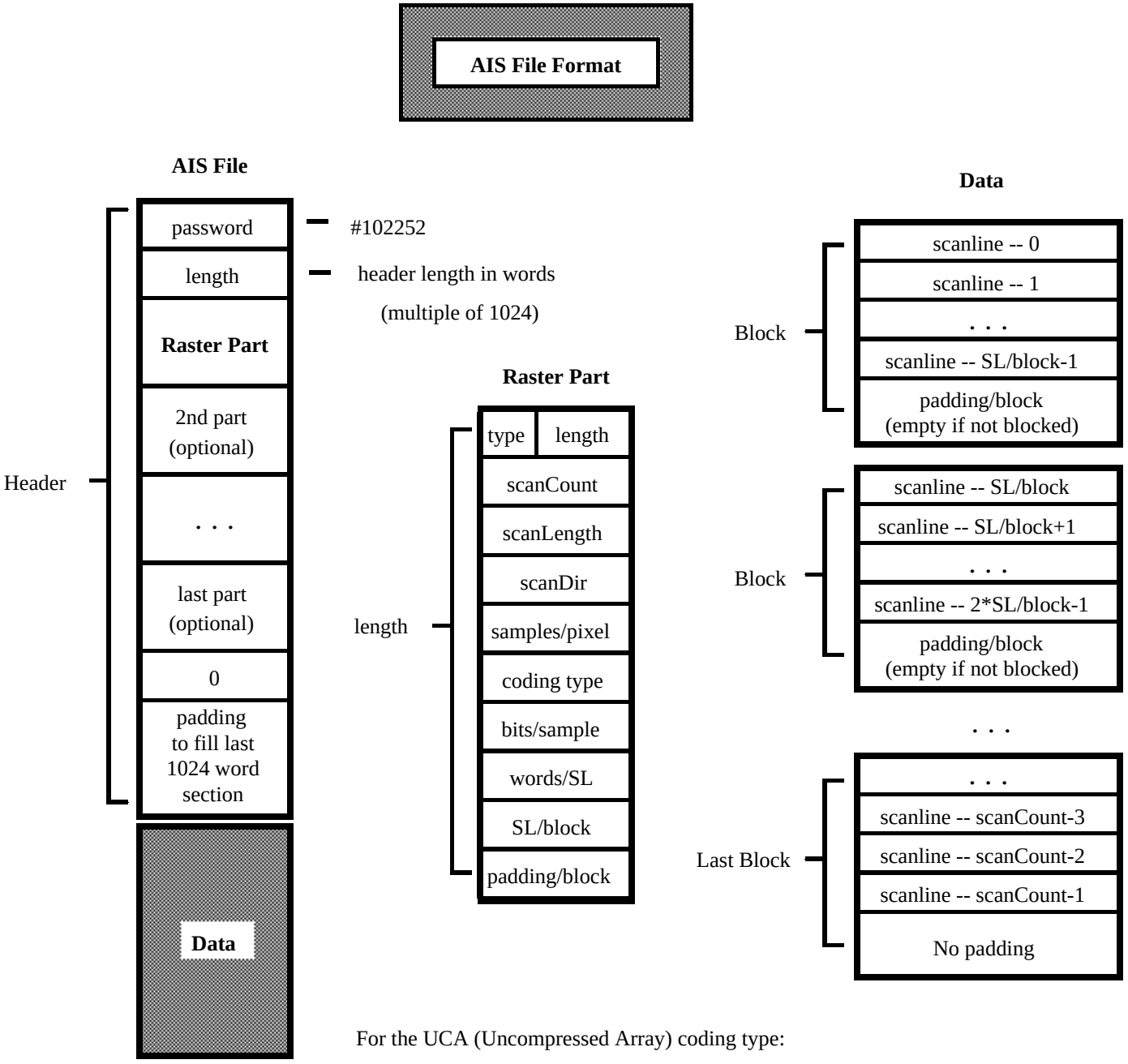


Figure 4. .cu Font Format



For the UCA (Uncompressed Array) coding type:

A block may be an arbitrary length.

Numerically it is equal to: (words/SL)*(SL/block)+(padding/block)

The Data length is: (Nblocks*Lblock)+(words/SL)*(scanCount rem SL/block)

where Nblocks=scanCount/(SL/block) and Lblock is the block length.

If the file is unblocked then the Data length is: scanCount*(words/SL)

Figure 5. Format of an AIS image file