ROS Utility Guide

RIDGE

9053

ROS Utility Guide

Ridge Computers

Santa Clara, CA

first edition: 9053 (SEP 84)

PUBLICATION HISTORY

Manual Title: ROS Utility Guide

first editon: 9053 (SEP 84)

NOTICE

ACKNOWLEDGEMENT

This software and documentation is based in part on the fourth Berkeley Software Distribution, under license from the regents of the University of California. We acknowledge the following individuals for their part in its development: Ken Arnold, Rick Blau, Earl Cohen, Robert Corbett, John Foderaro, Mark Horton, Bill Joy, Randy King, Jim Kleckner, Steve Levine, Colin McMaster, Geoffrey Peck, Rob Pike, Eric Scott, and Eric Shienbrood.

These tutorial guides are based on documents created at Bell Laboratories to describe UNIX System software, and at the University of California, Berkeley, to describe bsd 4.2. Credits are given on the first page of each document contained in this volume.

Some text has been changed to more accurately describe Ridge Computers' implementation of the software. Inappropriate material may have been deleted.

UNIX is a trademark of Bell Laboratories.

PREFACE

The ROS Utility Guide (manual 9053) is a collection of tutorial documents related to software utility programs. Each section contains the detailed information that is omitted from the page of the same name in the ROS Reference Manual (9010).

The topics in the Table of Contents are not the only ones related to utility programs. The ROS Reference Manual has many entries that are fully explained within. Typically, the user will see a program in the ROS Reference Manual, and if one of these tutorials is mentioned under the **SEE ALSO** heading, he/she will turn to this Utility Guide for help. After the reader is familiar with a topic, he/she might refer to the ROS Reference Manual only.

TABLE OF CONTENTS

Ridge Multi-Window Display Management

This document describes the Ridge Operating System (ROS) multi-window display management software for the Ridge Monochrome Display, and the integration of mouse software with the bit-map display hardware.

See the **ROS Reference Manual** (9010) for related software mentioned in this document: setfont(1), settek(1), setx3.64(1), copybits(3X), graf(3X), wgraf(3X), windows(3X), font(4), mouse(7).

This document assumes the reader is familiar with the contents of the **Ridge Mouse Installation Manual and User's Reference (9042)**, which describes installation of a Ridge Mouse and its use with the Ridge Window Manager software.

1. INTRODUCTION

The Ridge multi-window display management software uses a graphics display *screen* for text and graphics output, and the *keyboard* and *mouse* pointing device for input, to substantially increase the efficiency of interaction with the Ridge 32.

The display management software supports client processes via standard ROS system calls that do not require any special knowledge of the multi-window environment. The typical user process directs its terminal input and output to the display using standard "read" and "write" interfaces, and the display management software takes care of multiplexing the input/output to the associated window for each user process. The window management functions are invoked through existing standard ROS system calls so that the interface is language- and application-independent. The window-specific functions allow applications to build their own interactive environments that can run concurrently with applications executing in other windows on the same display.

The display management software consists of several system processes that provide the interactive user interface to the rest of the ROS system. The *Display Driver* process manages the keyboard input and the multiple window output for a single Ridge Monochrome Display. The Display Driver manages multiple windows that may overlap on the screen, with the contents of each window being updated concurrently, thus allowing the user to view the progress of multiple activities at the same time.

Only one window may be enabled for input at any one time; switching between windows is done with the press of a mouse button, thus allowing the user to easily switch from one activity to another.

The Display Driver maintains a collection of data structures associated with each window. New windows are created and manipulated by application programs through standard system calls. By making appropriate input, output, and control requests to the Display Driver, a group of processes can treat a window as a *virtual terminal* that has attributes and status independent from the other windows on the display screen.

The *Mouse Manager* process reads the mouse input and sends reports of the current position and the button status to the Display Driver. User programs access the mouse input information through the Display Driver, which associates both the mouse and keyboard input with the active input window.

The *Window Manager* is an application program that provides the interactive user interface to the Display Driver via the mouse device. The Window Manager interprets mouse input to allow selection of the current input window, and performs other window manipulations such as creation of new windows or placement of windows on the screen via the menu selection capability. Each window created by the Window Manager is associated with its own shell command interpreter process, which can be used to invoke other programs.

2. ENVIRONMENT

It is important to understand the features of the Ridge Monochrome Display, keyboard, and mouse devices.

2.1. Ridge Monochrome Display

The Ridge Monochrome Display is a *bit-mapped raster video* output device. Its hardware controller board generates a video image from an on-board 128K-byte refresh memory. The memory data is bitmap that controls the color (black or white) of each dot on the screen. The refresh memory is updated by DMA transfers from the main memory that is accessible to the CPU.

Individual bits in contiguous refresh memory are mapped directly to the picture elements *(pixels)* on the screen. Both text and graphics are created by software that sets the appropriate bits in the display memory, thus allowing arbitrary image generation.

The resolution of the screen is 1024 pixels in the X (horizontal) direction, and 800 pixels in the Y (vertical) direction. Display device coordinates are expressed in terms of the pixel addresses, with the origin (0, 0) placed at the upper left corner of the screen. The individual pixels are addressed from 0 to 1023 in the X direction, and 0 to 799 in the Y direction.

The bitmap memory is a dot-matrix representation of the rectangular image. A pixel is addressable as a bit in the display memory. By default, a bit value zero corresponds to a white pixel, and a bit value one corresponds to a black pixel. A register in the display controller hardware may be set to invert this correspondence, so that zero corresponds to black and one corresponds to a white pixel.

For maximum flexibility and low software overhead, each user process can access the bitmap in its own address space using standard Ridge LOAD and STORE machine instructions. It is possible for each process to change the screen image by modifying a bitmap in its own data space, which, because of the specific location in the virtual address space, is mapped to the display controller refresh memory as backing store by the virtual memory system.

The bitmap occupies 128K contiguous bytes in the virtual address space of each process. The virtual memory system copies 4096-byte pages of the bitmap in and out of the display controller refresh memory.

Modification of the screen image is achieved by modifying bits in a virtual address space, and by use of ROS Kernel primitive Flush to send modified pages in main memory to the display controller refresh memory. Main memory pages that are mapped to the display may be reclaimed for other use by the virtual memory system if they have not been modified recently.

2.2. Keyboard

The display management software is controlled by keyboard input. Each time a key is pressed, an 8-bit character is generated. The alphanumeric and punctuation keys generate standard ASCII codes, while the special function keys and cursor control keys generate single 8-bit characters that have the most significant bit set. The non-ASCII special key codes are described in part 6: *Ridge Keyboard Codes* of this document.

2.3. Mouse

The mouse is a pointing device with three buttons that slides on a reflective surface near the display. It sends a sends a serially-encoded *report* when its position or buttons change.

Mouse movements are translated into cursor motion on the screen. In this way, the mouse is a device for pointing at objects on the screen. Normally, mouse movements are translated into cursor motions at a ratio of 1:2. A single incremental movement of the mouse, however, translates to a single pixel cursor movement for fine positioning control. Thus, the mouse can move rapidly over a large screen area and maintain the precision to select a particular spot.

The three mouse buttons have two states ("pressed" and "released"), giving eight possible combinations. The software, however, treats each button independently. Some mouse-controlled functions are invoked by *dragging* the mouse; pressing a button and holding it down while the mouse is moved. Other functions are invoked with a *click* of the button, by pressing the button and immediately releasing it.

3. DISPLAY DRIVER

Any process may access a display bitmap in its own virtual address space, but it is more efficient for one bitmap to be manipulated by multiple processes. The *Display Driver* process provides the coordination to allow multi-process access to one bitmap. There exists one Display Driver process per display device on the system.

The Display Driver software supports different screen sizes and resolutions to allow for future display configurations. The resolution of the screen can be determined from a user process by making an *ioctl(2)* system call using a file descriptor that is associated with the Display Driver:

```
#include <termio.h>

struct DisplayInfo {
        int width;
        int height;
        unsigned int *bitmap;
};

ioctl (filedesc, command, arg)
        struct DisplayInfo *arg;
```

The *command* argument using this form is:

BCGETSZ Get the parameters associated with the display and store them in the DisplayInfo structure referenced by *arg*.

The *width* and *height* of the display are measured in bits. The *bitmap* field will contain the base address in virtual memory that corresponds to the 128K-byte display bitmap.

3.1. Windows

A *window* (or *viewport* in computer literature) is the primary mechanism used to share the screen. Each window on the screen is a rectangular region that may overlap other windows. Each window provides a "task environment" that is separate from the other windows on the screen.

Text and graphics displayed in a window are clipped to the window's boundaries. When the window frame gets smaller, or text scrolls past the window boundaries, the text not displayed in the window is no longer retrievable. Graphics are automatically scaled relative to the size of a window, so that no graphical information is lost when the size of a window is changed.

A window appears as a rectangle with a 2-pixel-wide outline and a *title tab* above the upper left corner containing its title string. Windows appear in a shade that distinguishes them from the background pattern. Windows may overlap each other. Each window maintains its place in a list that is sorted by the logical depth ordering. The first window in the list may overlap and obscure any window that is behind it in the list.

The Display Driver maintains a separate set of "retained graphics" or "display list"data to represent the textual and graphical operations that have been performed within each window. When a portion of the display must be recreated due to movement of windows, the Display Driver automatically redraws the affected part of the screen based on information from the data structures. A user process has no access to these structures.

The Display Driver also maintains information for each window that allows it to act as a "virtual terminal" to a set of processes. Each window has a state associated with it to control features such as echoing of input characters, canonical or raw input mode, interpretation of special characters such as the INTR or QUIT characters, and control over the interpretation of sequences of ASCII characters that are output to the window. The state and attributes of each window can be queried and modified via the *ioctl(2)* system call independently from other windows.

For instance, a window may be set in a mode to interpret escape sequences for cursor positioning and line/character editing that are compatible with the ANSI X3.64 standard, thus allowing the window to emulate terminals such as the DEC VT100. The particular escape sequences and control codes supported are described later in this document.

Alternatively, a window may be set to interpret output characters in a mode that is compatible with the Tektronix 4014 graphics terminal, thus providing line-drawing graphics output. Tektronix 4014 and ANSI X3.64 output codes are described in the part 7 of this section.

3.2. Window Creation

Processes normally inherit their association with a window from their parent process (typically the shell) when they are created. Subsequent terminal input/output for each process is directed to the appropriate window, with the Display Driver performing the bitmap operations necessary to actually display text and graphics on the screen.

New windows may be created using either the *creat(2)* or *open(2)* system calls, providing the name of the window as the last part of a pathname that specifies the display device. The display device is accessed by the pathname */dev/dispN* where the "N" stands for a specific device number, as described in *disp(7)*.

The *window name* is appended to the display device name, separated from it by a slash "/". For example, to create a new window whose name is "window2" on display device number "0", the pathname used with *creat(2)* would be "/dev/disp0/window2".

Processes which inherit their control terminal from their parent, which is the normal case, or processes which explicitly set their control terminal, are associated with a particular window on a display. The special name */dev/tty* identifies the window corresponding to the process' control terminal. This special name is mapped by ROS to the actual pathname of the window. Thus, processes may easily refer to their control window without having to know the actual window pathname.

When the Display Driver process is started, it automatically creates a single window, whose size is the entire display screen. This window, which is called the *prototype window*, has a null window name, so that it is referred to simply by the display device name */dev/dispN*, where the "N" is the specific display number. Whenever input or output is directed to the prototype window, the Display Driver automatically makes the prototype window the active window. This handling of the prototype window allows commands like *write(1)* and *wall(1)*, to send messages to the displays without knowing specific window names.

Once a new window has been created, or an existing window opened, a process directs its input and output to the window using standard ROS system calls such as *read(2)* and *write(2)*, or library routines built on top of these basic system calls. Output causes appropriate changes to the rectangular area on the screen associated with the window, where the changes depend on the current mode of the window as described later. Input requests allow the process to read the keyboard input associated with the window. Non-blocking reads may be used by setting the O_NDELAY flag either with *open(2)* or with *fcntl(2)*. Access to the mouse input is described part 3.8 of this document.

When all file descriptors associated with a window have been closed in all processes, via *close(2)*, the Display Driver kills the window and removes it from the screen. A window may be forcibly destroyed via the *unlink(2)* system call; any outstanding system calls pertaining to the deleted window will return errors to the caller.

3.3. Window Control Functions

Various window attributes and operations are accessed by the *ioctl(2)* system call. For example, the terminal characteristics described in *termio(7)* are accessed for each window independently from the other windows on the display.

The *windows(3)* subroutine package allows access to the various window-specific control functions by providing a simplified interface that performs the appropriate *ioctl* call for each function. These functions are described in the following sections; file descriptor 2 *(stderr)* is used to perform the *ioctl* request, and the value -1 is returned as an error indicator.

Upon creation, each window is assigned a *window ID* that identifies it in all subsequent control operations.

```
int GetWindowNumber (filedesc)
int filedesc;
```

returns the window ID of the window associated with the open file descriptor argument. The "prototype window" is assigned a window ID with the value 0. A value of -1 indicates that the file descriptor is not associated with a window on the display.

```
int GetCurrentWindow ()
```

returns the window ID of the active input window, which is always at the front of the depth-sorted window list.

```
int GetNextWindow (wID)
int wID;
```

returns the window ID of the window which is next on the window list behind the given window. The value -1 is returned if there are no more windows farther back in the list.

Window size and position on the screen are described using the *Point* and *Rectangle* data structures, as defined in the *<sys/graf.h>* header file. A Point is a two-dimensional location specified in terms of display device coordinate values.

```
struct Point {
                int x;
                int y;
};
```

A Rectangle is specified as two Points, where the origin specifies the upper left corner location, and the extent specifies a width and height measured in bits.

```
struct Rectangle {
                struct Point origin;
                struct Point extent;
};
```

```
GetWindowFrame (wID, frame)
int wID;
struct Rectangle *frame;
```

determines the current location and size of a window's rectangular frame. When a window is created, its origin is (0,0) and its extent is (0,0).

```
SetWindowFrame (wID, frame)
int wID;
struct Rectangle *frame;
```

moves the frame of a window on the screen, and can change its size.

```
int FindWindow (at)
struct Point *at;
```

determines the frontmost window that contains a given point on the display.

If a window can be found that contains the point either within its frame or title tab, then its window ID is returned; otherwise, the value -1 is returned. The location of the point is specified using display device coordinates.

When a window is created, the window part of the pathname is copied to another string which is used as the text for displaying the title tab. The title string may be read, or changed to another string, by the following functions.

```
char *GetTitle (wID)
int wID;

SetTitle (wID, name)
int wID;
char *name;
```

GetTitle returns a pointer to a static area containing a null-terminated string, which should be copied elsewhere by the caller. Changing the title string does not change the name of the window. For example, a window that was created with the name "/dev/disp0/window2" originally has the title "window2"; changing the title to "box" causes the title tab to change, but the window is still referred to by the name it was created with. Changing a title to the null string causes the title tab to disappear.

Each window has a set of flags, which are represented as bits within a 32-bit *wFlags* word. The bits control various attributes of the window, including the emulation mode of the window.

```
int GetWFlags (wID)
int wID;
```

determines the *WFlags*.

```
SetWFlags (wID, flags)
int wID;
int flags;
```

modifies the *wFlags*.

The following bits, or bit fields, are defined in the *<sys/winctrl.h>* header file.

WFMode Bit field that determines how sequences of characters are interpreted for a window. The following three mutually exclusive modes are currently defined.

WFASCII Mode that performs no interpretation of escape sequences, treating output characters as standard ASCII characters. Characters are displayed using the font associated with the window. The only control characters interpreted are Backspace, Linefeed, Return, and Bell, which produce their normal functions (Bell flashes the screen momentarily).

WFANSIX3_64
Mode that interprets control characters and escape sequences compatible with ANSI X3.64 standard. This is the default mode, set when a window is created.

WFTek4014
Mode that interprets control characters and output sequences compatible with the Tektronix 4014 graphics terminal.

WFAwake Bit that determines if a window is "awake". The initial value of this bit is 1, allowing input and output. If the bit is cleared to 0, then the current size of the window and its associated text and graphics is saved, and the window is put to "sleep". The window is cleared and all input. Output requests are held pending until the window is awakened. When the bit is then set back to 1, the window attains the previously-saved size, regardless of size changes since the window was put to sleep, and any text or graphics are redrawn.

WFCurson Bit that determines if an inverted-block cursor appears in a window at the current text cursor location. This bit is initially set to 1, causing the text cursor to appear whenever an outstanding input request is pending in the active input window. If this bit is cleared to 0, then no text cursor is displayed in the window.

WFRetainGraf
Bit that determines if graphics operations are retained for a window operating in Tektronix 4014 mode. This bit is initially set to 1, causing all graphics operations to be saved in an internal data structure so that the image can be regenerated as needed. If this bit is cleared to 0, then graphics operations are not retained.

```
SelectWindow (wID)
int wID;
```

selects a window as the active input window, and moves it to the front of the depth-sorted window list. The selected window is displayed in front of all other windows, and its title tab is highlighted. All keyboard and mouse input events are directed to the active input window.

```
UnderWindow (wID)
int wID;
```

moves the indicated window behind all awake windows in the depth-sorted window list. This causes the new frontmost window in the list to become the active window.

Certain events such as INTR or QUIT keystrokes in a window cause signals to be sent to all processes that have the window as their control terminal. The control window for a process is inherited from its parent process when it is created.

```
SetCtrlWindow (wID)
int wID;
```

changes the window which acts as the control terminal for the calling process.

```
KillWindow (wID)
int wID;
```

sends the SIGHUP signal to all processes which have the given window as their control terminal. The SIGHUP signal causes all the processes associated with the window to terminate, possibly after cleaning up temporary data.

3.4. Text Fonts

A *font* is a sequence of pixel patterns corresponding to the images of individual characters. These sequences of bit matrices are kept in files normally found in the */fonts* directory in the ROS file system. The various font files contain different patterns corresponding to different type styles and sizes. The format of these files is described in *font(4)*.

The size of a bit-matrix font corresponds, roughly, to printer's *points* (72 points to the inch). By convention, the names of the font files include the point size. For example, *fix10* contains a typeface whose size is approximately 10 points.

The characters in the *fix* fonts have the same bit-matrix size so that text in a window can be arranged into rows and columns of the same length, as on a regular terminal, which is essential for cursor positioning commands. The Display Driver reads the */fonts/sys.font* file at initialization time, and uses this font information as the default font for depicting characters. The contents of the */fonts/sys.font* file should be the same as one of the other *fix* font files.

The image of characters can also be represented as a sequence of vectors, or line segments. Then, by appropriate scaling, the same font can be used to generate many different type sizes. This alternative encoding scheme for fonts is used by the Display Driver to depict text as line drawings in a window that is in the Tektronix 4014 graphics emulation mode. The Display Driver reads the */fonts/sys.vfont* file at initialization time to obtain this vector font information.

Text is displayed in a window according to the window's font. Each window has a single font associated with it, but different windows can each have their own font. The default font is associated with a newly created window, but may be changed on a per window basis using the *setfont(1)* command. When a window's font is changed, all text in the window is redisplayed using the new font information.

```
int DefineFont (name)
char *name;
```

makes a font known to the Display Driver.

The parameter is a the full pathname of a bit-matrix font file. The Display Driver reads the contents of the font file into an internal data structure, and returns a font ID that is used in further operations on that font. The value -1 is returned if the file cannot be read, or if it does not contain font information.

```
char *GetFontName (fontID)
int fontID;
```

returns a pointer to a null-terminated string that is the pathname that was used to read a given font ID. The pointer should be copied to another area by the caller.

```
int GetFontID (wID)
int wID;

SetFontID (wID, fontID)
int wID;
int fontID;
```

determines or sets the font ID associated with a window.

```
GetWCharSize (wID, charSize)
int wID;
struct Point *charSize;
```

determines the width and height of a single character in the bit-matrix font currently associated with the given window.

The size of the character returned is measured in bits. This may be used to calculate the number of lines and columns of text that can be shown in a window.

3.5. ANSI X3.64 Compatibility Mode

When a new window is created, it is initialized to the ANSI X3.64 compatible mode. In this mode, the window can emulate terminals such as the DEC VT-100 that comply with the ANSI X3.64 standard. The number of lines and columns of text depends both on the size of the window and on the size of the bit-matrix font associated with the window.

Printable ASCII characters are displayed using the bit-matrix font associated with the window. Non-printable characters are shown as a small "lightning bolt" which indicates there is no corresponding bit pattern for that 8-bit character code. Certain control characters, such as Return, Linefeed, and Backspace, perform their customary cursor control functions.

Output characters are displayed at the current position of the text cursor for a window. The text cursor is shown as a video-inverted block the size of one character position whenever the window has an outstanding input request. The text cursor is advanced to the right one position as each character is written, and automatically moves to the first column of the next line after the last character in a line is written.

Various escape sequences allow cursor positioning, insertion and deletion of characters and lines, clearing parts of lines and windows, and inverse video on a per-character basis. The particular escape sequences that are recognized in X3.64 mode are described later in this document.

A user can set the X3.64 mode in a window by executing the *setx3.64(1)* command in that window. The X3.64 mode can be set from a program by either modifying the *wFlags* word associated with the window, or by writing a particular escape sequence that is recognized in any mode, setting X3.64 mode. The four-character escape sequence is **ESC %! 1**

3.6. Tektronix 4014 Compatibility Mode

A window may be set to the Tektronix 4014 mode. This allows emulation of the Tektronix 401X series of graphics terminals on the Ridge Monochrome Display, except where hardware differences make it infeasible to do so. Line-drawing graphics are supported through commands embedded in the output data, which are compatible with the Tektronix 4014 equipped with the Enhanced Graphics Module features.

Graphics and text are clipped to the window boundaries, and are scaled proportionately to the difference in resolution between a real Tektronix 4014 display and the current window size. If the window is redrawn (such as when a window is moved or its size is changed), the graphics are automatically regenerated by the Display Driver.

In alphanumeric mode, printable ASCII characters are displayed using the vector font. The number of lines and columns of text is determined by the current character size, of which there are four to choose from. Non-printable characters are not displayed, and do not move the alphanumeric cursor. The Return, Linefeed, Backspace, and Vertical Tab control characters perform their customary cursor control functions.

In graphics mode, character sequences define endpoints of lines to be drawn. The endpoints are specified in a coordinate range of 0 to 4095 in both the X and Y directions, which are mapped to actual display device coordinates using a scaling factor that is proportional to the current size of the window. A line may be drawn in one of five styles: solid, dotted, dot-dashed, short-dashed, and long-dashed. Point Plot Mode, Incremental Plot Mode, and Graphics Input Mode are supported. The details of the Tektronix 4014 emulation mode are described later in this document.

A set of simple subroutines for line drawing is described in *wgraf(3)*. In addition to generating the proper character sequences to specify a line segment or to clear the window, there are functions to control the width and color of lines. By default, lines are drawn one pixel wide, but it is possible to increase the width in one-pixel increments. Also by default, lines are drawn as black pixels, but either white pixels or the complement of the current background pixels may be specified.

A user can set the Tektronix 4014 mode in a window by executing the *settek(1)* command in that window. The Tektronix 4014 mode can be set from a program by either modifying the *wFlags* word associated with the window, or by writing a particular escape sequence that is recognized in any mode, setting Tektronix 4014 mode. The four-character escape sequence is **ESC %! 0**

When a window is placed into Tektronix 4014 mode, alphanumeric mode is selected, the large character size is set, the line style is set to solid, the window is cleared, and the alphanumeric cursor is placed at the home position.

3.7. Keyboard Input

Keyboard interrupts are handled by the Display Driver, and queued in the type-ahead buffer of the active input window. Control over the queueing of characters is described later in this document. The interpretation of special characters such as INTR and QUIT is controlled on a per-window basis as described in *termio(7)*.

A process uses the standard *read(2)* system call to access the keyboard input for the window associated with an open file descriptor. Normally, a process blocks when it attempts to read from a window that has no keystrokes queued for it, or if the window is not the active input window. A non-blocking *read(2)* may be used to determine if there are any characters from the keyboard queued for a particular window. If the O_NDELAY bit is set either by *open(2)* or *fcntl(2)* on a file descriptor bound to a window, then a *read* returns immediately with a value of 0 when there are no keystrokes queued for the window.

3.8. Mouse Input

The *Mouse Manager* is a process separate from the Display Driver which handles a mouse or similar pointing device interfaced through an RS-232 serial port. The Mouse Manager is normally started as an asynchronous process during the login procedure, as described later.

The Mouse Manager reads the input device data and converts the data into a canonical form of X and Y positions and the current button states. This information is sent to the Display Driver, which places the information coded as a sequence of bytes into the mouse event queue for the active input window. Control over the queueing of mouse events is described later in this document.

The Display Driver places 5 bytes in the event queue for each mouse event. The entire event queue is flushed first if there is not enough room left for all 5 bytes, thus insuring that only the most recent events are queued.

The first byte contains the state of the buttons. Bit 0 (least significant) is the right button, bit 1 is the middle button, and bit 2 (more significant) is the left button. The other bits are normally zero, but may contain more button bits if a nonstandard pointing device is being used. A bit value of 1 indicates the corresponding button is depressed, and a bit value of 0 indicates the button is released.

The second and third bytes contain the X coordinate, in two's complement representation, high-order byte first. The fourth and fifth bytes contain the Y coordinate in the same format. The coordinate system used for the X,Y position depends on the mode of the active window and the event mode bits in the *wFlags* word for the window.

If the EMLocCoords bit is set, then display device coordinates are used, and the location can be anywhere on the display surface. The X coordinates range from 0 to 1023, and the Y coordinates range from 0 to 799, with the origin in the upper left corner of the Ridge Monochrome Display *disp(7)*.

If the EMLocCoords bit is not set, only mouse positions within the active window are queued. The coordinate system used in this case is translated relative to the location of the window on the display surface, and may be scaled depending on the current mode of the window.

If the window is emulating a Tektronix 4014 terminal (WFMode is WFTek4014), then the X coordinates range from 0 to 1024, and the Y coordinates range from 0 to 780, with the origin in the lower left corner of the window. The display device coordinates of the mouse are appropriately scaled based on the size of the window.

If the window is not in WFTek4014 mode, then display device coordinates are used, but are first translated relative to the origin of the window. The X coordinates range from 0 to one less than the width of the window, and the Y coordinates range from 0 to one less than the height of the window, with the origin in the upper left corner of the window.

The mouse input for a window is accessed using the *open(2)* system call, providing the name "mouse" appended to the window pathname, separated from it by a slash "/". For example, to open the mouse input associated with window "window2" on display device number "0", the pathname would be "/dev/disp0/window2/mouse".

Alternatively, the special name */dev/mouse* can be used by a process to specify the mouse input from the window which corresponds to its control terminal. This special name is mapped by ROS to the actual pathname of the window plus the string "/mouse". Thus, processes may refer to the mouse input from their control window without having to know the actual window pathname.

A process uses the standard *read(2)* system call to access the mouse input from the window associated with an open file descriptor. Each Read returns as many bytes from the event queue as are requested, but to maintain synchronization with the queuing of mouse event bytes, it is recommended that 5 bytes be read at a time.

Normally, a process blocks when it attempts to read from a window that has no mouse input queued for it, or if the window is not the active input window. A non-blocking *read(2)* may be used to determine if there is any mouse input queued for a particular window. If the O_NDELAY bit is set either by *open(2)* or *fcntl(2)* on a file descriptor bound to a window, then a read returns immediately with a value of 0 when there are no mouse input bytes queued for the window.

3.9. Input Event Handling

Several of the bits in the *wFlags* word associated with each window provide control over the queueing of input events from the keyboard and mouse. The *wFlags* are determined and modified using the *GetWFlags* and *SetWFlags* functions described previously.

The keyboard events for the active input window are controlled by the following bits, which are defined in the <*sys/winctrl.h*> header file.

EMQueueKB
: Bit that determines if any keystrokes are to be queued for the window. The initial value of this bit is 1, allowing normal interpretation of keyboard interrupts, including special characters such as INTR and QUIT. If this bit is cleared to 0, then all keyboard interrupts for the window are ignored.

EMSigIOKB
: Bit that determines if keystrokes cause a SIGIO *signal(2)* to be sent to the process or process group waiting for keyboard input from the window. The initial value of this bit is 0, causing no signals to be sent. If this bit is set to 1, then each keyboard interrupt for the window generates a signal to the processes associated by *fcntl(2)* with the window.

Mouse input is placed in the mouse event queue for the active input window according to the following bits, as defined in the <*sys/winctrl.h*> header file. If the bits for the active window do not allow queueing of a mouse event, then the window associated with the Window Manager process is tested according to its bits. This allows a process in the active window to filter some or all mouse events, passing unwanted events to the Window Manager, which may choose to handle the mouse event or have it discarded.

EMQueueLoc
: Bit that determines if mouse input is to be queued for the window. The initial value of this bit is 0, causing all mouse events for the window to be ignored. If this bit is set to 1, then mouse events which satisfy the conditions specified by the other control bits are queued.

EMLocCoords
: Bit that determines the allowable coordinates of the mouse which are queued for the window. The initial value of this bit is 0, allowing only mouse events whose coordinates are inside the window's boundaries to be queued. If this bit is set to 1, then mouse events located anywhere on the screen which satisfy the other conditions are queued.

EMLocMotion
: Bit that allows queueing of mouse events resulting from motion or button changes. The initial value of this bit is 0, allowing only mouse events that indicate a button change to be queued, thus ignoring events resulting only from motion. If this bit is set to 1, then mouse events are queued which satisfy the other conditions, regardless of button changes.

EMButtonMask
: Bit field that determines which button-changes will cause a mouse input event to be queued for the window. The field is composed of the logical OR of the following

four fields, and is initialized to the value 0.

EMRightButton
: Bit that corresponds to the right button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the right button from the previous event is to be queued. A bit value of 0 indicates that right button changes are to be ignored.

EMMiddleButton
: Bit that corresponds to the middle button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the middle button from the previous event is to be queued. A bit value of 0 indicates that middle button changes are to be ignored.

EMLeftButton
: Bit that corresponds to the left button on the mouse. A bit value of 1 indicates that a mouse event which has a different state for the left button from the previous event is to be queued. A bit value of 0 indicates that left button changes are to be ignored.

EMOtherButtons
: Bit that corresponds to all other buttons on a nonstandard mouse or other pointing device. A bit value of 1 indicates that a mouse event which has a different state for at least one of the other buttons from the previous event is to be queued. A bit value of 0 indicates that all other button changes are to be ignored.

EMSigIOLoc
: Bit that determines if mouse input events cause a SIGIO *signal(2)* to be sent to the process or process group waiting for mouse input from the window. The initial value of this bit is 0, causing no signals to be sent. If this bit is set to 1, then each mouse event (not each byte) for the window generates a signal to the processes associated by *fcntl(2)* with the window.

The *wFlags* bits that control mouse input events may also be accessed by making an *ioctl(2)* system call using a file descriptor associated with a window.

```
#include <termio.h>

ioctl (filedesc, command, arg)
int *arg;
```

The *command* argument using this form is

MOUSEGET
: Get the flag bits associated with the mouse input into the integer pointed to by *arg*. All other bits are cleared to zero.

```
ioctl (filedesc, command, arg)
int arg;
```

The *command* argument using this form is

MOUSESET
: Set the flag bits associated with the mouse input from *arg*. All other bits are ignored.

3.10. Cursor

The *cursor* is a small block of pixels that is displayed on the screen at the current location corresponding to the mouse, and thus tracks the movement of the mouse. The cursor location is normally constrained to remain within the boundaries of the display screen by the Display Driver process.

The normal shape of the cursor is an arrow which points upward and to the left. The tip of the arrow indicates the pixel that corresponds to the current mouse location.

When a window emulating a Tektronix 4014 graphics terminal is placed into Graphics Input (GIN) mode by a special character sequence, the shape of the cursor changes to a cross-hair. The intersection of the cross-hair indicates the pixel that corresponds to the current mouse location. During GIN mode, the cursor may also be moved in single-pixel increments by the four cursor keys on the keyboard (labelled with left, right, up, and down arrows). Any other keystroke terminates GIN mode in the active input window.

```
SetCursorLocation (wID, at)
int wID;
struct Point *at;
```

sets the current cursor location relative to a given window.

The coordinate system used for the X,Y position depends on the mode of the given window and the event mode bits in the *wFlags* word for the window.

If the EMLocCoords bit is set, then display device coordinates are used, and the location can be set to any value. If the EMLocCoords bit is not set, then the coordinate system is translated relative to the location of the window on the display surface, and may be scaled depending on the current mode of the window.

If the window is emulating a Tektronix 4014 terminal (WFMode is WFTek4014), then the X coordinates range from 0 to 1024, and the Y coordinates range from 0 to 780, with the origin in the lower left corner of the window. The display device coordinates of the cursor are appropriately scaled based on the size of the window.

If the window is not in WFTek4014 mode, then display device coordinates are used, but are first translated relative to the origin of the window. The X coordinates range from 0 to one less than the width of the window, and the Y coordinates range from 0 to one less than the height of the window, with the origin in the upper left corner of the window.

3.11. Pop-up Menus

Pop-up menus are supported by the Display Driver. A pop-up menu displays a list of commands on the screen and allows the user to point to the desired command, selecting it with a mouse button. When a mouse button is pressed and held down, a menu appears at the location of the cursor. The pop-up menu overlays part of the screen image, which is saved before the menu appears, and is restored when the menu selection is complete.

The user invokes a pop-up menu by pressing a button, holding the button down while moving the mouse to select the desired line of text in the menu, and then releasing the button. As the mouse is moved with the button down, the selected menu line is highlighted in inverse video. If the button is released while the mouse is outside the pop-up menu boundary, no operation is performed.

By convention, application programs dedicate the middle and right mouse buttons to be used for selecting commands from one of two menus. The menu associated with the middle button contains commands that pertain to the application running in the active input window. This menu may be non-existent for a particular window, depending on the currently executing application. The menu associated with the right button contains commands relevant to the Window Manager, that is, commands that apply to the display as a whole, such as opening, closing, or moving a window.

An application process creates a pop-up menu by sending a string of text and a button indicator to the Display Driver. The text, consisting of several short lines, is displayed outlined by a rectangle at the current position of the screen cursor. When one of the lines is selected using the mouse button, the index of the line is returned to the calling program, which can perform the appropriate function.

```
int PopupMenu (text, button)
char *text;
int button;
```

The *text* parameter points to a null-terminated string, with embedded Newline characters separating each line. The *button* parameter is a bit mask in the lowest three bits, with the least significant bit representing the right button, the middle bit the middle button, and the most significant bit the left button. The menu is shown on the screen, and selection is allowed, while a button is depressed which has the corresponding bit set to 1. When the button is released, the selected line's index (counting from one) is returned. If the button is released outside of the menu, the value zero is returned.

4. WINDOW MANAGER

The *Window Manager* process provides an interactive user interface to the window functions supported by the Display Driver. It allows the user to create and destroy windows, position them on the screen or change their size, modify the depth order of overlapping windows, and select the active input window. The Window Manager is controlled by the mouse, and provides its commands in a pop-up menu that may be invoked anywhere on the screen by pressing a button on the mouse.

The Window Manager is a regular user process that is normally started during the initial login sequence as part of the user's *.profile* or *.login* commands. An empty screen with only the gray background is presented when the Window Manager first runs. A Window Manager will kill itself if its *stderr* is not associated with a display, or if another Window Manager is already running on that display.

When the Window Manager starts, it changes the size of the "prototype window" to a small rectangle in the center of the screen, then puts it to sleep, making it disappear. If input or output is directed specifically to the prototype window, the Display Driver awakens it automatically, causing it to appear in front of other windows, with no title tab. The prototype window may be hidden again by using either the "sleep" or "close" commands described below.

The Window Manager waits for mouse events and, based on the buttons, either performs a window operation selected from a pop-up menu, or selects a window to become the active input window.

A click of the left button selects the frontmost window containing the screen cursor as the new current window. The selected window is brought to the front of all other windows, and its title tab is highlighted.

The middle button, which is not used by the Window Manager, is dedicated to application-specific functions in each different window.

The right button invokes a pop-up menu for window manipulation. To select a command from the pop-up menu, press the right button and hold it down. A different menu command is highlighted as you slide the mouse vertically. Release the right button when the desired menu command is highlighted. The following Window Manager commands are available from the pop-up menu:

Open creates a new window, and starts a new shell process in it. An outline of a window with 24 rows and 80 columns of characters appears on the screen with its upper left corner tracking the cursor. When the right button is clicked, the window is drawn at its current location. The newly-created window becomes the selected active input window, placed in front of all other windows with its title tab highlighted. A shell process is created and its *stdin, stdout,* and *stderr* are bound to the new window. The shell to be executed is found from the SHELL environment variable if it exists, or from the shell field in the user's *passwd(4)* entry if it exists, or finally using */bin/sh* if the previous methods fail.

Close	kills all the processes associated with the current window, thus causing the window to disappear. The processes are sent the SIGHUP signal, causing them to terminate their activities, possibly after cleaning up their working environment.
Move	drags the current window's border around on the screen, following the mouse, until the right button is clicked. The window is redrawn at the new location.
Grow	drags the lower right corner of the current window's border around on the screen following the mouse, while the upper left corner stays fixed. When the right button is clicked, the window is redrawn with the new size.
Under	places the current window behind all the other windows (except ones that are asleep), automatically selecting the next window in front as the new current window.
Sleep	shrinks the current window to a small size, and suspends its input and output. Selecting a sleeping window twice with the left button causes it to wake up, resuming its previous size and re-enabling its input and output.
Exit	causes the Window Manager to exit, thus returning back to a blank screen and the *login* process. This may only be done when there are no more windows open on the screen.

5. INSTALLING MULTI-WINDOW SOFTWARE

Read **Ridge Mouse Installation Manual and User's Reference (9042)** for installation basics. This document provides additional technical tips on installation.

The Mouse Manager associated with a display, by default, reads the serial device **/dex/tty***x* whose *X* number corresponds to the **/dev/disp***X*. For example, the Mouse Manager started on */dev/disp0* reads its input from */dev/tty0*. If a command line parameter is given when the Mouse Manager is started, the Mouse Manager instead uses the specified pathname as its input device. Use of the command line parameter is recommended in **Ridge Mouse Installation and User's Guide** (9042) because it is more manageable.

The Mouse Manager uses its *stdout* file descriptor to associate itself with the Display Driver, The Mouse Manager kills itself if the *stdout* is not bound to a Display Driver, or if another Mouse Manager is already associated with that Display Driver. The Mouse Manager must have exclusive use of its serial device, so the file */etc/inittab* must not cause a User Monitor process or the *getty(1)* process to be started for that */dev/tty* device.

Starting the Mouse Manager is accomplished from the shell by executing the program */ros/mousemgr*, optionally passing a device pathname as an argument, and specifying that the program run as an asynchronous process with an ampersand "&" at the end of the command line. As described below, this command is generally placed into the *.profile* or *.login* command file for a user.

The Window Manager is invoked from the shell as a simple command */usr/bin/windowmgr*, with no arguments. The Window Manager uses its *stderr* file descriptor to associate itself with the Display Driver, and will kill itself if either the *stderr* is not bound to a Display Driver, or if another Window Manager is already associated with that Display Driver.

6. RIDGE KEYBOARD CODES

The following 8-bit hexadecimal codes are generated by the non-ASCII-character keys on the Ridge keyboard. The labels "S1" through "S8" refer to the unlabelled "soft" keys at the top of the keyboard, ordered from left to right.

Key label	8-bit code
F1	80
F2	81
F3	82
F4	8D
F5	8E
F6	8F
F7	93
F8	94
F9	95
F10	99
F11	9A
F12	9B
F13	9F
F14	A0
F15	A1
S1	83
S2	84
S3	85
S4	86
S5	87
S6	88
S7	89
S8	E2
PREV	8A
LINE INSERT	8B
CHAR INSERT	8C
NEXT	90
LINE DELETE	91
CHAR DELETE	92
\	96
↑	97
/	98
←	9C
HOME	9D
→	9E
/	A2
↓	A3
\	A4

7. TERMINAL EMULATION MODES

The Ridge Monochrome Display operates in either Tektronix 4014^{tm} emulation mode, or in ANSI X3.64 mode. Unless otherwise directed, the display operates in X3.64 mode.

If the Ridge Window Manager software is in use, references in this manual to the "screen" or "Display" apply to each individual window.

The mode of operation is controlled by software. It can be set by typing a control sequence at the Display keyboard or by executing a ROS command.

MODE	*SEQUENCE*	*COMMAND*
Tektronix	ESC %! 0	settek(1)
ANSI X3.64	ESC %! 1	setx3.64(1)

7.1. ANSI X3.64 Mode for Monochrome Display

7.1.1. Control Characters

Control characters are interpreted in X3.64 mode as follows:

NUL and **DEL**: ignored on output
CR: moves the cursor to first column in current line
LF and **FF**: moves the cursor down one line, scrolling if at last line
BS: moves cursor one column to left unless already at left edge
BEL: causes screen to "flash"

7.1.2. Escape Sequences

In the tables that follow, "ESC" stands for the escape key (hexadecimal code 1B), and "n" stands for a decimal numeric parameter in the range 0 to 9. If omitted, "n" assumes the value 0 or 1, depending on the function.

ANSI X3.64 escape sequences

NAME		
Position cursor	ESC [n ; n H	**Position cursor at the line specified by first parameter n, and to the column specified by the second parameter n. Lines and columns are numbered starting from 1, which is the default value of n. The terminator "f" is equivalent to "H".**
Cursor up	ESC [n A	Move cursor up n lines (n defaults to 1).
Cursor down	ESC [n B	Move cursor down n lines (n defaults to 1).
Cursor right	ESC [n C	Move cursor right n columns (n defaults to 1).
Cursor left	ESC [n D	Move cursor left n columns (n defaults to 1).
Index	ESC D	Move cursor down one line. If cursor was on last line, scroll screen upward.
Reverse index	ESC M	Move cursor up one line. If cursor was on first line, scroll screen downward.
Next line	ESC E	Move cursor to the beginning of the next line. If cursor was on last line, scroll the contents of the screen upward.
Erase to end of line	ESC [0 K	Erase characters in current line from the cursor to the end of the line. If no numeric parameter is given with the "K" terminator, 0 is the default.

ANSI X3.64 escape sequences (continued)

NAME		
Erase from beginning of line	ESC [1 K	**Erase characters in current line from the beginning up to, but not including, the cursor position.**
Erase line	ESC [2 K	Erase all characters in the line with the cursor.
Erase to end of screen	ESC [0 J	Erase characters in current line from the cursor to the end of the screen. If no parameter is given with the "J" terminator, 0 is the default.
Erase from beginning of screen	ESC [1 J	Erase characters from the beginning of the screen up to, but not including, the cursor position.
Erase screen	ESC [2 J	Erase entire screen.
Insert line	ESC [n L	Insert n new lines (n defaults to 1). The line containing the cursor and any following lines are scrolled downwards.
Delete line	ESC [n M	Delete n lines (n defaults to 1) starting with the line containing the cursor. Any following lines are scrolled upwards.
Delete character	ESC [n P	Delete n characters (n defaults to 1) starting with the character under the cursor. Any remaining characters are slid over to the left with blanks filling on the right end.
Insert character mode	ESC [4 h	Enter insert mode, where each printed character causes characters from the cursor to be slid over to the right before the character is written.
Replace character mode	ESC [4 l	Exit insert mode; each character is written over any character at the position of the cursor (default mode).
Set video attributes	ESC [0 m	Set default video attributes (reset inverse video). If no parameter is given with the "m" terminator, 0 is the default.
Select inverse video	ESC [7 m	Set inverse video attribute for any following printing characters, until attributes are turned off.

7.2. Tektronix 4014 Emulation Mode for Monochrome Display

All emulation functions are performed in software. Any program written for the Tektronix 4014tm series terminals will run in this emulation mode.

See the *Tektronix 4014 Computer Display Terminal User's Manual.*

The following Tektronix features are available:

- Enhanced Graphics Module features
  (4096 addressibility, Point Plot Mode, Incremental Mode)
- X-Y addressing of 0-4095
  mapped into a 1024 x 780 logical display area
  (which is then scaled and displayed in the window)
- Five line types
  (solid, dotted, dot-dashed, short-dashed, long-dashed)
- Four character sizes
  (scaled to current window dimensions)

Five Tektronix modes are available:

- Alphanumeric Mode (with single margin control),
- Graphics Mode,
- Graphics Input Mode (GIN),
- Point Plot Mode, and
- Incremental Plot Mode.

The Ridge F1 key is equivalent to the Tektronix PAGE key. It selects Alpha mode, homes the alpha cursor, and erases the screen.

The Ridge F2 key is equivalent to the Tektronix RESET key. It selects Alpha mode, homes the alpha cursor and resets default characteristics, but does not erase the screen.

The following Tektronix 4014 features are not available on the Ridge Monochrome Display: write-thru, hard copy, audible bell, and intensity reduction.

7.2.1. Alphanumeric Mode Features

The following ASCII control characters move the alphanumeric cursor:

CR	moves the cursor to first column in current line
LF	moves the cursor down one line, wrapping around to the top if at last line
VT	moves the cursor up one line, wrapping around to the bottom if at first line
BS	moves cursor one column to left unless already at left edge

The following escape sequences result in various character sizes:

Escape sequence	Characters per line	Number of lines
ESC 8	35	74
ESC 9	38	81
ESC :	58	121
ESC ;	64	133

The Alphanumeric Mode features that are not supported are the Alternate Character Set, and Margin 2.

7.2.2. Graphics Mode Features

Graphics Mode is entered with GS or ESC GS sequence, and supports the Enhanced Graphics Module features. The Special Point Plotting feature is not supported. Graphics Mode is terminated when any of the following character sequences are output: ESC FF, ESC US, US, ESC SUB, RS, ESC RS, FS, or ESC FS.

Five line styles are provided: solid, dotted, dot-dashed, short-dashed, and long-dashed. The first pixel of a line in any style will always be drawn. Line styles are selected with the following escape sequences, consisting of an Escape character followed by any one of the listed characters:

Escape Sequence	Line Style
ESC ',e,f,g,h,m,n,o,p,u,v,w	Solid
ESC a,i,q	Dotted
ESC b,j,r	Dot-Dash
ESC c,k,s	Short-Dash
ESC d,l,t	Long-Dash

Lines are specified as a sequence of characters that define their endpoints, in the following standard order:

HIY EXTRA (optional) LOY HIX LOX

"Shortened addresses" may be used, where only the bytes which change need to be sent. The emulation saves the values for HIY, LOY, and HIX, and draws the vector when the LOX byte is received. The following table describes which bytes must be sent if a specific byte is changed.

Changed	Bytes which must be received				
	HIY	EXTRA	LOY	HIX	LOX
HIY	*		*		*
EXTRA		*	*		*
LOY			*		*
HIX			*	*	*
LOX					*

7.2.3. Graphics Input Mode Features

ESC SUB Enters the window into graphics input mode (GIN) and causes a crosshair to be displayed. The cursor may be positioned by the mouse or the four arrow keys (left, right, up, and down) located on the right side of the keyboard.

ESC ENQ Received when the window is in Graph Mode causes transmission of the Terminal Status Byte followed by the 4-byte address of the Graph Mode cursor position.

ESC ENQ Received when the window is in Alpha Mode causes transmission of the Terminal Status Byte followed by the 4-byte address of the Alpha Mode cursor position.

ESC ENQ Received when the window is in GIN Mode causes transmission of the 4-byte address of the crosshair cursor. The window returns to Alpha Mode upon completion of transmission.

Any key other than the cursor keys that is typed when the crosshair is displayed will result in the transmission of that character, followed by the 4-byte address of the crosshair position.

The Terminal Status Byte is defined as follows:

Bit 7:	0 always
Bit 6:	0 always
Bit 5:	1 always
Bit 4:	1 = no hardcopy attached
Bit 3:	0 = vector move, 1 = vector draw
Bit 2:	0 = Graph Mode, 1 = Alpha Mode
Bit 1:	0 = No margin 2
Bit 0:	1 = No auxiliary device attached

7.2.4. Point Plot Mode

Point Plot Mode is entered by sending FS or ESC FS when in Alpha or Graph Modes. The data for drawing is identical to normal graphics, but the output consists of only the final end-point of the lines.

7.2.5. Incremental Plot Mode

Incremental plot mode is entered by sending RS or ESC RS when in Alpha or Graph Modes. After RS is sent, either SP (pen up) or P (pen down) must be sent. One point increments then are plotted (or moved to) in the direction which is dependent on the ASCII characters as follows:

Character	X-direction	Y-direction
D	0	+1
E	+1	+1
A	+1	0
I	+1	-1
H	0	-1
J	-1	-1
B	-1	0
F	-1	+1

UNIX-to-UNIX Copy (UUCP) Network

SECTION 1
UUCP CONCEPTS

INTRODUCTION

The **uucp** network allows information exchange between UNIX systems over the public telephone system. This section discusses concepts and explains the use of the user level interface to the network. Understanding these basic principles helps the user make the best possible use of the **uucp** network.

Some major uses of the network are:

- distribution of software
- distribution of documentation
- personal communication (mail)
- data transfer between closely sited machines
- transmission of debugging dumps and data exposing bugs
- production of hard copy output on remote printers.

THE UUCP NETWORK

The **uucp**(1) network is a network of UNIX systems that allows file transfer and remote execution of programs. The extent of the network is a function of both the interconnection hardware and the controlling network software. Membership in the network is tightly controlled by the software to preserve the integrity of all members of the network. You cannot use the **uucp** facility to send files to systems that are not part of the **uucp** network.

Network Hardware

The **uucp** was originally designed as a dialup network so that systems in the network could use the public telephone system to communicate with each other. The three most common methods of connecting systems are:

1. Connecting two UNIX systems directly by cross-coupling (via a null MODEM) two of the computers ports. This means of connection is useful for only short distances (up to several hundred feet can be achieved, although the RS-232 standard specifies a shorter distance) and is usually run at high speed (9600 or 19200 baud). These connections run on asynchronous terminal ports.

2. Using a MODEM (a private line or a limited distance MODEM) to directly connect processors over a private line.

3. Connecting a processor to another system through a MODEM, an automatic calling unit (ACU), and the public telephone system. This is the most common interconnection method, and it makes the largest number of connections available. Modems with integral automatic dialers eliminate the need for a separate ACU.

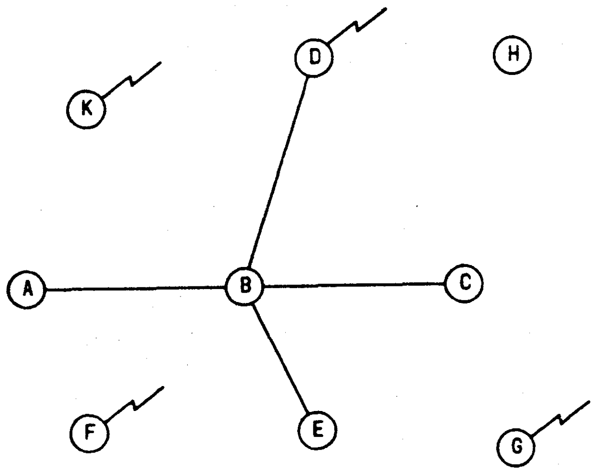
The **uucp** could be extended to use higher speed media (e.g., HYPERchannel*, Ethernet†, etc.), and this possibility is being explored for future UNIX system releases.

Network Topology

Connections between systems are possible via the public telephone system. The topology of the network is determined by both the hardware connections and the software that control the network.

Hardware Topology

As discussed earlier, it is possible to build a network using permanent or dial-up connections. In Figure 1, a group of systems (A, B, C, D, and E) are shown connected by hard-wired lines. All systems are assumed to have some answer-only data sets so that remote users or systems can be connected.



* Trademark of Network Systems Corporation. † Trademark of Xerox Corporation.

= automatic calling unit or auto-dial MODEM
= computer system

Figure 1. Uucp Nodes

K, D, F, and G have automatic calling units or auto-dial MODEMs and H has no capability for calling other systems. Users should be aware that the network consists of a series of point-to-point connections (A-B, B-C, D-B, E-B) even though it appears in Figure 1 that A and C are directly connected through B. The following observations are made:

1. System H is isolated. It can be made part of the network by arranging for other systems to poll it at fixed intervals. This is an important concept to remember because transfers from systems that are polled do not leave the system until that system is called by a polling system.

2. System K, F, G, and D easily reach all other systems because they have calling units.

3. If system A (E or G) wishes to send a file to H (K, F, or G), it must first send it to D (via system B) because D is the only system with a calling unit.

Software Topology

The hardware capability of systems in the network defines the maximum number of connections in the network. The software at each node restricts the access by other systems and thereby defines the extent of the network. The systems of Figure 1 can be configured so that they appear as a network of systems that have equal access to each other or some restrictions can be applied. As part of the security mechanism used by **uucp**, the extent of access that other systems have can be controlled at each node. Figures 2 and 3 show how the network might appear at one node.

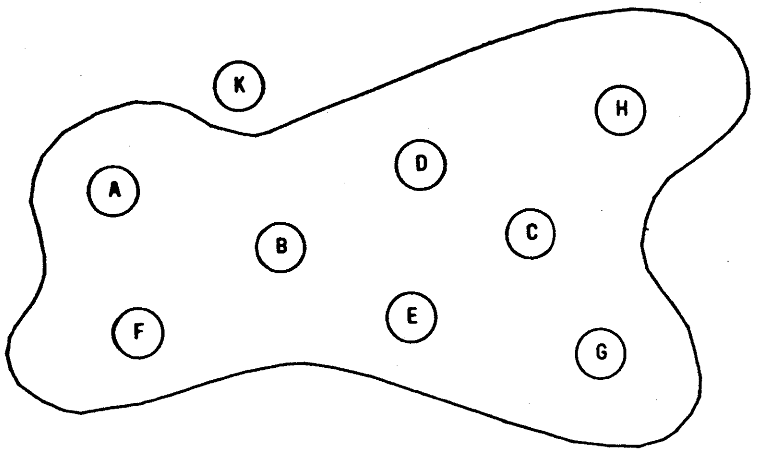


Figure 2. Uucp Network Excluding One Node

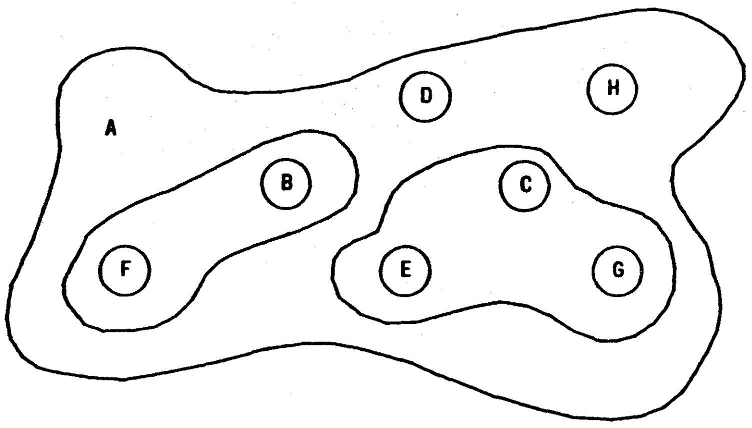


Figure 3. Uucp Network with Several Levels of Permissions

Access is available from all systems in Figure 2. In Figure 3, however, some of the systems have been configured to have more or less access privileges than others (systems C, E, and G have one set of access privileges, systems F and B have another set, etc.).

The **uucp** uses the UNIX system password mechanism coupled with a system file (**/usr/lib/uucp/L.sys**) and a file system permission file (**/usr/lib/uucp/USERFILE**) to control access between systems. The password file entries for **uucp** (usually, **luucp, nuucp, uucp,** etc.) allow only those remote systems that know the passwords for these IDs to access the local system. (Great care should be taken in revealing the password for these **uucp** logins because knowing the password allows a system to join the network.) The system file (**/usr/lib/uucp/L.sys**) defines the remote systems that a local host knows about. This file contains all information needed for a local host to contact a remote system (including system name, password, login sequence, etc.) and as such is protected from viewing by ordinary users.

In summary, while the available hardware on a network of systems determines the connectivity of the systems, the combination of password file entries and the **uucp** system files determine the extent of the network.

Security

A most critical feature of any network is the security that it provides. Users are familiar with the security that the UNIX system provides in protecting files from access by other users and in accessing the system via passwords. In building a network of processors, the notion of security is widened because access by a wider community of users is granted. Access is granted on a system basis (that is, access is granted to all users on a remote system). This follows from the fact that the process of exchanging a file with another system is done by daemons that use one special user ID(s). This user ID(s) is granted (denied) access to the system by the **uucp** system file (**/usr/lib/uucp/L.sys**) and the areas that the system has access to is controlled by another file (**/usr/lib/uucp/USERFILE**). For example, access can be granted to the entire file system tree or limited to specific areas.

Software Structure

The **uucp** network is a batch network. When a request is made, it is spooled for later transmission by a daemon. This is important to users because the success or failure of a command is only known at some later time by **mail**(1) notification. For most transfers, there is little trouble in transmitting files between systems, but transmissions are occasionally delayed or fail because a remote system cannot be reached.

Transfer Rules

There are several rules by which the network runs. These rules are necessary to provide the smooth flow of data between systems and to prevent duplicate transmissions and lost jobs. The following paragraphs outline these rules and their influence on the network.

Queuing

Jobs submitted to the network are assigned a sequence number for transmission. Jobs are represented by a file (or files) in a common spool directory (**/usr/spool/uucp**). When a file transfer daemon (**uucico**) is started to transmit a job, it selects a system to contact and then transmits all jobs to the system. Before breaking off the conversation, any jobs to be received from that remote system are accepted. **Uucp** may be sending to or receiving from many systems simultaneously. The number of incoming requests is only limited by the number of connections on the system, and the number of outgoing transfers is limited by the number of auto-dial MODEMs, ACUs, or direct connections.

Dialing and the Public Telephone System

In order to transfer data between processors that are not directly connected, an auto-dialer is used to contact the remote system. There are several factors that can make contacting a remote system difficult.

1. All lines to the remote system may be busy. There is a mechanism within **uucp** that restricts contact with a remote system to certain times of the day (week) to minimize this problem.
2. The remote system may be down.
3. There may be difficulty in dialing the number (especially if a large sequence of numbers involving access through PBXs is involved).

Scheduling and Polling

When a job is submitted to the network, an attempt to contact that system is made immediately. Only one conversation at a time can exist between the same two systems.

Systems that are polled can do nothing to force immediate transmission of data. Jobs will only be transmitted when the system is polled (hourly, daily, etc.) by a remote system.

Retransmissions and Fixed Delay

The **uucp** network attempts to contact remote systems to complete a transmission. To prevent **uucp** from continually calling systems that are unavailable, there exists a minimum fixed delay (currently 55 minutes) before another transmission can take place to that system.

Purging and Cleanup

Transfers that cannot be completed after a defined period of time (72 hours is the value that is set when the system is distributed) are deleted and the user is notified.

Special Places: The Public Area

In order to allow the transfer of files to a system on which a user does not have a login name, the **public** directory (kept in **/usr/spool/uucppublic**) is available with general access privileges. When receiving files in the **public** area, the user should dispose of them quickly as the administrative portion of **uucp** purges this area on a regular basis.

Permissions

File Level Protection

In transferring files between systems, users should make sure that the destination area is writable by **uucp**. The **uucp** daemons preserve execute permission between systems and assign "read write" permission (0666) to transferred files.

System Level Protection

The system administrator at each site determines the global access permissions for that processor. Thus, access between systems may be confined by the administrator to only some sections of the file system.

NETWORK USAGE

The following sections discuss the user interface to the network and give examples of command usage.

Name Space

In order to reference files on remote systems, they must be uniquely identified. The notation has several defaults to allow the reference to be compact. Some restrictions are placed on pathnames to prevent security violations. For example, pathnames may not include ".." as a component because it is difficult to determine whether the reference is to a restricted area.

Naming Conventions

Uucp uses a special syntax to build references to files on remote systems. The basic syntax is

```
system-name!pathname
```

where the system-name is a system that **uucp** is aware of. The *pathname* part of the name may contain any of the following:

1. A fully qualified *pathname* such as

   ```
   mhtsa!/usr/you/file
   ```

   The *pathname* may also be a directory name as in

   ```
   mhtsa!/usr/you/directory
   ```

2. The login directory on a remote may be specified by use of the ~ character. The combination ~user references the login directory of a user on the remote system. For example,

   ```
   mhtsa!~adm/file
   ```

   expands to

   ```
   mhtsa!/usr/sys/adm/file
   ```

   if the login directory for user adm on the remote system is **/usr/sys/adm.**

3. The *public* area is referenced by a similar use of the prefix ~/user preceding the pathname. For example,

   ```
   mhtsa!~/you/file
   ```

   expands to

   ```
   mhtsa!/usr/spool/uucppublic/you/file
   ```

   if **/usr/spool/uucp** is used as the spool directory.

4. Pathnames not using any of the combinations or prefixes discussed above are prefixed with the current directory (or the login directory on the remote). For example,

```
mhtsa!file
```

expands to

```
mhtsa!/usr/you/file
```

The naming convention can be used in reference to either the source or destination file names.

Forwarding

Although **uucp** does not allow specification of multiple sites, the **uusend(1)** command does.

Uusend sends a file to a given location on a remote system. The system need not be directly connected to the local system, but a chain of **uucp(1)** links needs to connect the two systems. For example:

```
uusend file hplabs! ucbbach! file
```

Types of Transfers

Uucp has a flexible command syntax for file transmission. The following are examples of different combinations of transfers.

Transmissions of Files to a Remote

Any number of files can be transferred to a remote system via **uucp**. The syntax supports the *, ? and [..] metacharacters. For example,

```
uucp *.[ch] mhtsa!dir
```

transfers all files whose name ends in c or h to the directory **dir** in the users login directory on mhtsa.

Forwarding may be done by **uusend**; otherwise the syntax is the same.

Fetching Files From a Remote

Files can be fetched from a remote system in a similar manner. For example,

```
uucp mhtsa!*.[ch] dir
```

fetches all files whose name ends in c or h from the users login directory on mhtsa and places the copies in the subdirectory *dir* on the local system.

Switching

Transmission of files can be arranged in such a way that the local system effectively acts as a switch. For example,

```
uucp mhtsb!files mhtsa!filed
```

fetchs files from the user's login directory on mhtsb, renames it as **filed**, and places it in the login directory on **mhtsa**.

Broadcasting

Broadcast capability (that is, copying a file to many systems) is not offered by **uucp**. But it can be simulated by a shell script:

```
for i in mhtsa mhtsb mhtsd
do
        uucp file $i!broad
done
```

Unfortunately, one **uucp** command is spawned for each transmission so that it is not possible to track the transfer as a single unit.

Remote Executions

The remote execution facility allows commands to be executed remotely. For example,

```
uux "!diff  mhtsa!/etc/passwd  mhtsd!/etc/passwd>!pass.diff"
```

executes the command **diff**(1) on the password file on mhtsa and mhtsd and places the result in **pass.diff**.

Spooling

Normally **uucp** copies its source files to the spool area before beginning transmission. This allows you to continue modifying a file without affecting the transmitted copy. You can specify to **uucp** that it should not do this copy by use of the **-c** option. For example, the following command will send the file **work** from the current directory when connection is established with **mhtsa**.

```
uucp -c work mhtsa!~/you/work
```

Notification

The success or failure of a transmission is reported to users via the **mail**(1) command. The choices for notification are:

1. Notification returned to the requesters system (via the **-m** option). This is useful when the requesting user is distributing files to other machines. Instead of logging onto the remote machine to read mail, mail is sent to the requester when the copy is finished.

2. **Uux**(1) always reports the exit status of the remote execution. Status information is appended to the **/usr/spool/uucp/LOGFILE** file.

Job Control

Jobs are controlled in the following ways.

Requeuing a Job

Uucp clears its working area of jobs regularly (usually every 72 hours), to prevent a buildup of jobs that cannot be delivered. The **-r** option forces the date of a job to be changed to the current date, thereby lengthening the time that **uucp** attempts to transmit the job. The **-r** option does not impart immortality to a job; it only postpones deleting the job during house-keeping functions until the next cleanup.

Network Names

Users may find the names of the systems on the network via the **uuname**(1) command. Only the names of the systems in the network are printed.

Network Status

Uulog(1) prints information about completed work done for a specific system or a specific user. **Uulog -u** *user* prints all information regarding work done for that user since the last time the **uucp** log file (/usr/spool/uucp/LOGFILE) was cleared. Similarly, **uulog -s** *system* prints all information regarding work done for that system.

Uusnap(1) prints information about pending work. For each system that either has files spooled or commands ready for remote execution, a line is printed. For example,

```
hplabs    2cmd  3data  2xqt   locked
ucbbach   1cmd  -----  ----
```

UTILITIES THAT USE UUCP

Several utilities rely on **uucp**(1) or **uux**(1) to transfer files to other systems.

Mail

The **mail**(1) command uses **uux** to forward mail to other systems. For example, when a user types

```
mail mhtsa!tom
```

the **mail** command invokes **uux** to execute **rmail** on the remote system (**rmail** is a link to the **mail** command). Forwarding mail through several systems (e.g., mail a!b!tom) is simulated by the **mail** command itself.

Netnews

The **netnews**(1) command that is locally supported on many systems uses **uux** in much the same way that **mail** does to broadcast network mail to systems subscribing to news categories. **Netnews**(1) is not currently supported by Ridge Computers.

SECTION 2
UUCP ADMINISTRATION

INTRODUCTION

This section describes how a **uucp** network is set up, the format of control files, and administrative procedures. Administrators should be familiar with the ROS Reference Manual pages (manual 9010) for each of the **uucp** related commands.

PLANNING

In setting up a network of UNIX systems, there are several considerations that should be taken into account *before* configuring each system on the network. The following parts attempt to outline the most important considerations.

Extent of the Network

Some basic decisions about access to processors in the network must be made before attempting to set up the configuration files. If an administrator has control over only one processor and an existing network is being joined, then the administrator must decide what level of access should be granted to other systems. The other members of the network must make a similar decision for the new system. The UNIX system password mechanism is used to grant access to other systems. The file **/usr/lib/uucp/USERFILE** restricts access by other systems to parts of the file system tree, and the file **/usr/lib/uucp/L.sys** on the local processor determines how many other systems on the network can be reached.

When setting up more than one processor, the administrator has control of a larger portion of the network and can make more decisions about the setup. For example, the network can be set up as a private network where only those machines under the direct control of the administrator can access each other. Granting no access to machines outside the network can be done if security is paramount; however, this is usually impractical. Very limited access can be granted to outside machines by each of the systems on the private network. Alternatively, access to/from the outside world can be confined to only one processor. This is frequently done to minimize the effort in keeping access information (passwords, phone numbers, login sequences, etc.) updated and to minimize the number of security holes for the private network.

Hardware and Line Speeds

There are two methods of interconnection for **uucp**(1):

1. Direct connection using a null MODEM.

2. Connection over the public telephone system.

In choosing hardware, the equipment used by other processors on the network must be considered. For example, if some systems on the network have only 103-type (300-baud) data sets, then communication with them is not possible unless the local system has a 300-baud data set connected to a calling unit. (Most data sets available on systems are 1200-baud.) If hardwired connections are to be used between systems, then the distance between systems must be considered since a null MODEM cannot be used when the systems are separated by more than several hundred feet. The limit for communication at 9600-baud is about 800 to 1000 feet. However, the RS232 specification only allows for less than 50 feet. Limited distance MODEMs must be used beyond 50 feet as noise on the lines becomes a problem.

Maintenance and Administration

There is a minimum amount of maintenance that must be provided on each system to keep the access files updated, to ensure that the network is running properly, and to locate line problems. When more than one system is involved, the job becomes more difficult because there are more files to update and because users are much less patient when failures occur between machines that are under local control.

UUCP SOFTWARE

Figure 4 illustrates the daemons used by the **uucp** network to communicate with other systems. The **uucp**(1) or **uux**(1) command queues users requests and spawns the **uucico** daemon to call another system. Figure 5 illustrates the structure of **uucico** and the tasks that it performs in communicating with another system. **Uucico** initiates the call to another system and performs the file transfer. On the receiving side, **uucico** is invoked to receive the transfer. Remote execution jobs are actually done by transferring a command file to the remote system and invoking a daemon (**uuxqt**) to execute that command file and return the results.

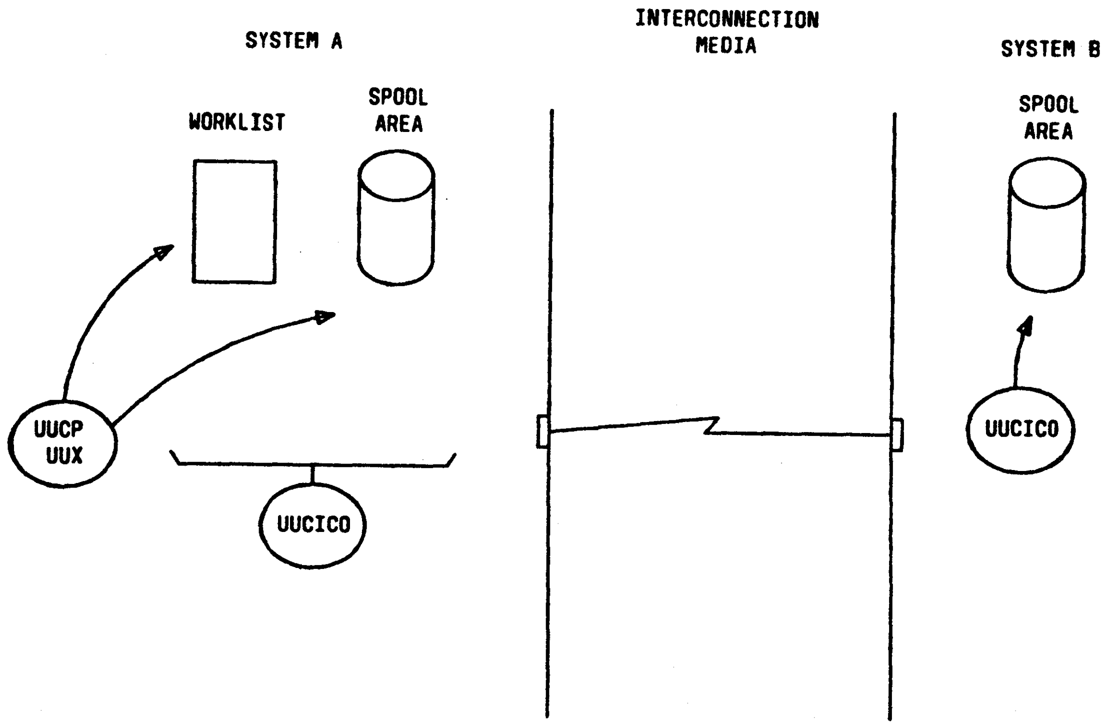


Figure 4. Uucp Network Daemon

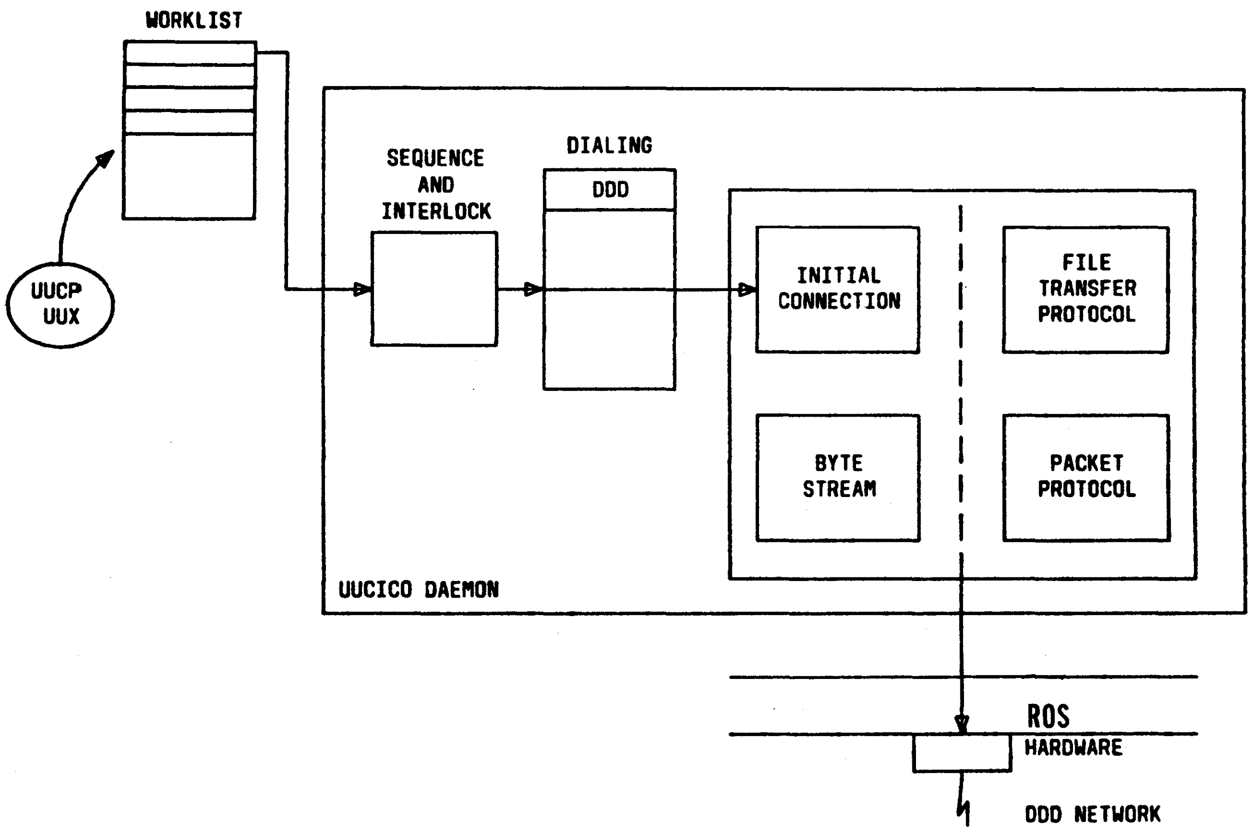


Figure 5. Uucico Daemon Functional Blocks

INSTALLATION

The **uucp**(1) package is delivered by Ridge Computers as a stand-alone package. The **uucp** package is initially installed by a Ridge Computers Systems Engineer. It is then the responsibility of the system administrator to modify the control files to reflect the specific configuration.

Executable Modules

The following executable modules are installed as part of the **uucp** installation procedure.

1. **Uucp** - file transfer command.
2. **Uulog** - command to print logfile summary information.
3. **Uux** - remote execution command.
4. **Uucico** - **uucp** network daemon.
5. **Uusnap** - network snapshot command.
6. **Uupoll** - command to initiate connection to a remote system.
7. **Uuclean** - cleanup command.
8. **Uusend** - command for sending a file across multiple hosts.
9. **Uuxqt** - remote execution daemon.
10. **Uuencode and uudecode** - commands to encode and decode a binary file for transmission via mail.
11. **Uuname** - print list of **uucp** systems.

In addition, several user-written shell scripts exist for maintaining a network. These include **uu.hourly, uucp.daily, uuq, uurate, uutbl, and uuusage.**

Password File (/etc/passwd)

To allow remote systems to call the local system, password entries must be made for any **uucp** logins. For example,

```
nuucp:zaaAA:6:1:UUCP.Admin:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

Note that the **uucico** daemon is used for the shell, and the spool directory is used as the working directory.

There must also be an entry in the **passwd** file for a **uucp** administrative login. This login is the owner of all the **uucp** object and spooled data files and must be "uucp". For example, the following is an entry in **/etc/passwd** for this administrative login:

```
uucp:zAvLCKp:5:1:UUCP.Admin:/usr/lib/uucp:
```

Note that the standard shell is used instead of **uucico**.

Lines File (/usr/lib/uucp/L-devices)

The file **/usr/lib/uucp/L-devices** contains the list of all lines that are directly connected to other systems or are available for calling other systems. The file contains the attributes of the lines and whether the line is a permanent connection or can call via a dialer. The format of the file is

```
type line call-device speed protocol
```

where each field is

type	If the line is directly connected to another system, the type is DIR. If the line uses an automatic calling unit or auto-dial MODEM the type is ACU.
line	The device name for the line (e.g., **tty1, tty6,** etc.).
call-device	If type is ACU, this is the device name of the ACU. Otherwise, the field is ignored; however, a placeholder must be used in this field so that the *protocol* field can be interpreted. Separate ACU units are not currently supported and this field must contain the word "**notused**".
speed	The line speed that the connection is to run at.
protocol	The type of auto-dial MODEM protocol to use.

Currently supported auto-dial MODEMs are:

hayes	smartmodem 300 or 1200
hayesq	similar to the hayes except MODEM is configured not to return result codes
vadic	3450 series
ventel	not specified
direct	direct connect

Examples of the file **/usr/lib/uucp/L-devices**:

```
DIR   tty3  unused  9600  direct
ACU   tty2  unused  1200  hayes
```

The first entry is for a hard-wired line running at 9600-baud between two systems. Note that the *acu-device* field is marked as unused. The second entry is for a line with a 1200-baud auto-dial MODEM.

System File (/usr/lib/uucp/L.sys)

Each entry in the **L.sys** file represents a system that can be called by the local **uucp** programs. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The fields are described below.

system name Name of the remote system.

time This is a string that indicates the days-of-week and times-of-day when the system should be called (e.g., MoTuTh0800-1730).

The day portion may be a list containing *Su, Mo, Tu, We, Th, Fr, Sa*; or it may be *Wk* for any week-day or *Any* for any day. The time should be a range of times (e.g., 0800-1230). If no time portion is specified, any time of day is assumed to be allowed for the call. Note that a time range that spans 0000 is permitted; 0800-0600 means all times are allowed other than times between 6 and 8 am. An optional subfield is available to specify the minimum time (minutes) before a retry following a failed attempt. The subfield separator is a "," (e.g., *Any,9* means call any time but wait at least 9 minutes before retrying the call after a failure has occurred). Several times may be or'ed together. For example, Any2300-0700| Sa| Su0000-1700, indicates any day of the week between 11 p.m. and 7 a.m., all day Saturday, or Sunday between midnight Saturday and 5 p.m.

type Connection type (ACU or DIR).

class This is the line speed for the call (e.g., 300) *phone* The phone number is made up of an optional alphabetic abbreviation (dialing prefix) and a numeric part. The abbreviation should be one that appears in the **L-dialcodes** file (e.g., mh1212, boston555-1212). For the hard-wired devices, this field contains the ACU or the hard-wired device name to be used for the call (e.g., tty0).

login The login information is given as a series of fields and subfields in the format

[*expect* | *send*] ...

expect is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect [-send-expect] ...

where the *send* is sent if the prior *expect* is *not* successfully read and the *expect* following the *send* is the next expected string. (For example, login--login will expect *login*; if it gets it, the program will go on to the next field; if it does not get *login*, it will send *null* followed by a new line, then expect *login* again.) If no characters are initially expected from the remote machine, the string "" (a null string) should be used in the first expect field.

There are several special names available to be sent during the login sequence. The string EOT sends an EOT character (control-d), and the string BREAK tries to send a BREAK sequence. The string PAUSEN causes the sequence to wait for 'n' seconds. The string LF sends a linefeed character; the string CR sends a carriage-return character. The string P-ZERO causes 8-bit characters to be sent with the parity bit always set to zero; the string P-ONE sets the parity bit to one; the strings P-EVEN and P-ODD cause even or odd parity generation, respectively.

There are several character strings that cause specific actions when they are a part of a string sent during the login sequence.

\s	Send a space character.
\d	Delay one second before sending or reading more characters.
\c	If at the end of a string, suppress the new-line that is normally sent. Ignored otherwise.
\b	Send a break signal.
\r	Send a carriage-return character.
\ddd	Send a character whose bit pattern is ddd, where ddd is one to three octal digits.

These character strings are useful for making **uucp** communicate over direct lines to data switches.

A typical entry in the **L.sys** file would be

```
sys Any ACU 300 mh7654 login uucp ssword: word
```

The "**expect**" algorithm matches all or part of the input string as illustrated in the password field above.

The following sequence illustrates most of the capabilities of **expect-send.** However, this sequence would better be replaced by specifying vadic auto-dialer protocol in L-devices.

```
rti-sel Any tty3 300 tty3 "" \05 *-\05-*
d NUMBER?-d-NUMBER? 5551212\r\d
LINE \r\d\r ogin: -\b-ogin: -\b-ogin:
Urti ssword: fatchance
```

means:

quote marks	expect nothing
\05	send control-e (activate auto-dialer)
-\05-	expect *, if not seen, send control-e, expect *
d	send a d to enter telephone number
NUMBER?-d-NUMBER?	look for prompt from auto-dialer, send d again if not seen
5551212\r\d	send telephone number, carriage return, pause one second, then send carriage return
LINE	wait till MODEM says on-line
\r\d\r	send return, pause, return
ogin: -\b-ogin: -\b-ogin:	look for login, if not found send *break*, look again, if still not found, send another *break*
Urti	send login ID urti
ssword	expect password:
fatchance	send password fatchance

Dialing Prefixes (L-dialcodes)

The **L-dialcodes** file contains those antiquated dial-code abbreviations used in the **L.sys** file (like AL for TW7-5678. The entry format is:

```
abb dial-seq
```

where *abb* is the abbreviation and *dial-seq* is the dial sequence to call that location.

The line

```
py 165-
```

would be set up so that entry py7777 would send 165-777 to the dial unit.

The **L-dialcodes** file is seldom used.

Userfile (USERFILE)

The **USERFILE** file contains user accessibility information. It specifies three types of constraints:

1. Files that can be accessed by a normal user of the local machine.

2. Files that can be accessed from a remote computer.
3. Login name used by a particular remote computer.

Each line in the file has the format

```
login,sys pathname [pathname] ...
```

where

login	is the login name for a user or the remote computer.
sys	is the system name for a remote computer.
pathname	is a pathname prefix that is acceptable for *sys*.

The constraints are:

1. When the program is obeying a command stored on the local machine, the pathnames allowed are those given on the first line in the USERFILE that has the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

2. When the program is responding to a command from a remote machine, the pathnames allowed are those given on the first line in the file that has the system name that matches the remote machine. If no such line is found, the first one with a *null* system name is used.

3. When a remote computer logs in, the login name that it uses *must* appear in the USERFILE. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

The line

```
u,m /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with **/usr/xyz**. The line

```
you, /usr/you
```

allows the ordinary user *you* to issue commands for files whose name starts with **/usr/you**. (This type restriction is seldom used.) The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows *any* remote machine to login with name *u*. If its system name is not *m*, it can only ask to transfer files whose names start with **/usr/spool**. If it is system *m*, it can send files from paths **/usr/xyz** as well as **/usr/spool**. The lines

```
root, /
, /usr
```

allow any user to transfer files beginning with **/usr** but the user with login *root* can transfer any file. (Note that any file that is to be transferred must be readable by anybody.) The line

```
,/
```

allows any user to transfer any files; it is very dangerous, but useful for systems with little need for security.

L.cmds

The file L.cmds specifies which commands the program **uuxqt** will execute on the local system. Typically it will restrict remote systems to execution of **rmail** (restricted mail) and **uusend** (for forwarding files). L.cmds can also allow any command to be executed, but this allows remote systems to have access to resources that perhaps they shouldn't have. A typical L.cmd follows:

```
# Example L.cmds
#
# This is the list of commands that uuxqt will execute.
# Default search path is /bin, then /usr/bin.
# It can be changed by a line starting with PATH:
PATH = /bin: /usr/bin: /usr/ucb
# The following commands are the only ones uuxqt will execute if the L.cmds file
is missing:
rmail
rnews
ruusend
# The following commands are potential security holes,
lpr
who
uusend
finger
```

ADMINISTRATION

The role of the **uucp** administrator depends heavily on the amount of traffic that enters or leaves a system and the quality of the connections that can be made to and from that system. For the average system, only a modest amount of traffic (100 to 200 files per day) pass through the system and little if any intervention with the **uucp** automatic cleanup functions is necessary. Systems that pass large numbers of files (200 to 10,000) may require more attention when problems occur. The following parts describe the routine administrative tasks that must be performed by the administrator or are automatically performed by the **uucp** package. The part on problems describes what are the most frequent problems and how to effectively deal with them.

Cleanup

The biggest problem in a dialup network like **uucp** is dealing with the backlog of jobs that cannot be transmitted to other systems. The following cleanup activities should be routinely performed by shell scripts started from **cron**(1).

Cleanup of Undeliverable Jobs

The **uucp.daily** procedure contains an invocation of the **uuclean** command to purge any jobs that are older than some fixed time (usually 72 hours). A similar procedure is usually used to purge any **lock** or **status** files. An example invocation of **uuclean**(1M) to remove both job files and old status files every 48 hours is:

```
/usr/lib/uucp/uuclean -pST -pC. -n48
```

Cleanup of the Public Area

In order to keep the local file system from overflowing when files are sent to the public area, the **uucp.daily** procedure can be set up with a **find** command to remove any files that are older than 7 days. This interval may need to be shortened if there is not sufficient space to devote to the public area.

Disposition of Log Files

The files SYSLOG and LOGFILE that contain logging information are saved by **uucp.daily** according to the following scheme,

LOGFILEs are daily moved to
    LOGFILE.day.month
where
    day is Sun, Mon, ..., Sat, and month is the numerical month of the year.

SYSLOG files are kept by

    SYSLOG.week (current week totals)
    SYSLOG.month (current month totals)
    SYSLOG.monthname (historical totals for that month, e.g. SYSLOG.Aug.)

Note that the **uucp.daily** shell script requires **uuusage** to generate the SYSLOG totals.

The **uucp.daily** script removes any LOGFILES older than three days. The script must be changed if you wish to save LOGFILES longer. SYSLOG files are never automatically removed, but do not usually represent nearly as much space as do LOGFILES.

Polling Other Systems

Systems that are passive members of the network must be polled by other systems in order for their files to be sent. This can be arranged by using the **uupoll** command as follows:

```
uupoll mhtsd
```

which will call **mhtsd** when it is invoked.

Problems

The following sections list the most frequent problems that appear on systems that make heavy use of **uucp**(1).

Out of Space

The file system used to spool incoming or outgoing jobs can run out of space and prevent jobs from being spawned or received from remote systems. The inability to receive jobs is the worse of the two conditions. When file space does become available, the system will be flooded with the backlog of traffic.

Bad MODEMs

MODEMs occasionally cause problems that make it difficult to contact other systems or to receive files. These problems are usually readily identifiable since LOGFILE entries will usually point to the bad line. If a bad line is suspected, it is useful to use the **cu**(1) command to try calling another system using the suspected line.

Administrative Problems

Some **uucp** networks have so many members that it is difficult to keep track of changing passwords, changing phone numbers, or changing logins on remote systems. This can be a very costly problem since MODEMs will be tied up calling a system that cannot be reached.

DEBUGGING

In order to verify that a system on the network can be contacted, the **uucico** daemon can be invoked from a user's terminal directly. For example, to verify that **mhtsd** can be contacted, a job would be queued for that system as follows:

```
uucp -r file mhtsd!~/tom
```

The **-r** option forces the job to be queued but does not invoke the daemon to process the job. The **uucico** command can then be invoked directly:

```
/usr/lib/uucp/uucico -r1 -x4 -smhtsd
```

The **-r1** indicates that the daemon is to start in master mode (i.e., it is the calling system). The **-x4** specifies the level of debugging that is to be printed. Higher levels of debugging can be printed (greater than 4) but requires familiarity with the internals of **uucico.** If several jobs are queued for the remote system, it is not possible to force **uucico** to send one particular job first. The contents of LOGFILE should also be monitored for any error indications that it posts. Frequently, problems can be isolated by examining the entries in LOGFILE associated with a particular system. The file ERRLOG also contains error indications.

SCCS – Source Code Control System

This document is based on a paper by Eric Allman of the University of California at Berkeley.

This document is an introduction to the Source Code Control System (SCCS). It is geared to programmers who want to accomplish a task, not those who want to know how it works on the inside. For this reason, some of the examples are not well explained. For details on the magic options, see the "Further Information" section.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file". There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes – equivalent to removing your changes later.

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form %*x*% where *x* is an upper case letter. For example, %I% is the SID of the latest delta applied, %W% includes the module name, SID, and a mark that makes it findable by a program, and %G% is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is "%W%" or whatever).

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin – i$i $i
        mv $i save/$i
end
```

This will put the named files into s-files in the subdirectory "SCCS" The files will be removed from the current directory and hidden away in the directory "save", so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory "save".

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print "No Id Keywords (cm7)", which is a warning message only.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

[2]Actually, the SID of the final delta applied was 1.1.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for "comments?" before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

---

[3]The "edit" command is equivalent to using the – e flag to *get*, as:

```
sccs get – e prog.c
```

Keep this in mind when reading other documentation.

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an "install" entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **– b** flag to ignore "branches" (alternate versions, described later) and the **– u** flag to only give files being edited by you. The **– u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info – ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get.* For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c    1.2    08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with "@(#)". This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c  1.2    08/29/80
/usr/bin/prog:
        prog.c  1.1    02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%        %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W%%G%";
```

in the file *opsys.h.* Otherwise, you will get compilation errors because "SccsId" is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID's consistent across files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit – r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit – r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get – r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the – **c** (cutoff) flag. For example,

```
sccs get – c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get – c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit – x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit – x1.3– 1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit – x1.3– 1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using – **x** (or – **i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of "a set of changes") can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prt command

When you created a delta, you presumably gave a reason for the delta to the "comments?" prompt. To print out these comments later, use:

```
sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2   80/08/29 12:35:31     bill    2     1     00005/00003/00084
removed "-q" option
D 1.1   79/02/05 00:19:31     eric    1     0     00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get – m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get – m – p prog.c | grep ˆ1.3´
```

The – **p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the "diff" flags can be used. To pass the – **c** flag, use – **C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The "deledit" command is equivalent to "delget" except that the "edit" command is used instead of the "get" command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix – r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The – r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The – d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs – d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file "/usr/src/cmd/SCCS/who.c". The file "who.c" will always be created in your current directory regardless of the value of the – d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[6]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get – k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin – z prog.c
```

11. Using the Admin Command

The *admin* command allows you to set parameters. The most interesting of these are flags. Flags can be added by using the – **f** flag. For example:

```
sccs admin – fd1 prog.c
```

sets the "d" flag to the value "1". This flag can be deleted by using:

```
sccs admin – dd prog.c
```

The most useful flags are:

b	Allow branches to be made using the – b flag to *edit*.
d*SID*	Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.
i	Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.
y	The "type" of the module. Actually, the value of this flag is unused by SCCS except that it replaces the *%Y%* keyword.

The – **t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the – **t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use "prt – t".

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

The ability to create branches must be enabled in advance using:

[6]Or given up and decided to start over.

```
sccs admin – fb prog.c
```

The – **fb** flag can be specified when the SCCS file is first created.

12.1. Creating a branch

To create a branch, use:

```
sccs edit – b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get – r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
sccs edit – i1.5.1.1– 1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs – d../xyz edit prog.c
```

When using the old version, be sure to use the – **b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info – b
```

when in the directory "xyz".

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the – **i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all cruft from the directory.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h

example: example.o
        $(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
        sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS=prog.c prog.h example.c

LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h

example: example.o
        $(CC) $(LDFLAGS) -o example example.o

sources: $(SRCS)
$(SRCS):
        sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say "make sources", and (3) the makefile is less likely to do confusing things since it won't try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS= a.o b.o c.o d.o
SRCS=a.c b.c c.c d.s x.h y.h z.h
TARG=        /usr/lib

# programs
GET= sccs get
REL=
AR=  - ar
RANLIB=      ranlib

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a

install: lib.a
        sccs check
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@

print: sources
        pr *.h *.[cs]
clean:
        rm - f *.o
        rm - f core a.out $(LIB)
```

The "$(REL)" in the get can be used to get old versions easily; for example:

```
make b.o REL=- r1.3
```

The *install* entry includes the line "sccs check" before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS= a.o b.o c.o d.o
SRCS=a.c b.c c.y d.s x.h y.h z.h

GET= sccs get
REL=

a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with "sccs". This list is not exhaustive; for more options see *Further Information.*

get	Gets files for compilation (not for editing). Id keywords are expanded.	
	– r*SID*	Version to get.
	– p	Send to standard output rather than to the actual file.
	– k	Don't expand id keywords.
	– i*list*	List of deltas to include.
	– x*list*	List of deltas to exclude.
	– m	Precede each line with SID of creating delta.
	– c*date*	Don't apply any deltas created after *date.*
edit	Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.	
	– r*SID*	Same as *get.* If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID.*
	– b	Create a branch.
	– i*list*	Same as *get.*
	– x*list*	Same as *get.*
delta	Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.	
unedit	Remove a file that has been edited previously without merging the changes into the s-file.	
prt	Produce a report of changes.	
	– t	Print the descriptive text.
	– e	Print (nearly) everything.
info	Give a list of all files being edited.	
	– b	Ignore branches.
	– u[*user*]	Ignore files not being edited by *user.*
check	Same as *info,* except that nothing is printed if nothing is being edited and exit status is returned.	
tell	Same as *info,* except that one line is produced per file being edited containing only the file name.	
clean	Remove all files that can be regenerated from the s-file.	
what	Find and print id keywords.	
admin	Create or set parameters on s-files.	
	– i*file*	Create, using *file* as the initial contents.
	– z	Rebuild the checksum in case the file has been trashed.
	– f*flag*	Turn on the *flag.*

– d*flag*	Turn off (delete) the *flag*.
– t*file*	Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.

Useful flags are:

b	Allow branches to be made using the – b flag to *edit*.
d*SID*	Default SID to be used on a *get* or *edit*.
i	Cause "No Id Keywords" error message to be a fatal error rather than a warning.
t	The module "type"; the value of this flag replaces the %Y% keyword.

fix	Remove a delta and reedit it.
delget	Do a *delta* followed by a *get*.
deledit	Do a *delta* followed by an *edit*.

2. Id Keywords

%Z%	Expands to "@(#)" for the *what* command to find.
%M%	The current module name, *e.g.*, "prog.c".
%I%	The highest SID applied.
%W%	A shorthand for "%Z%%M% <tab> %I%".
%G%	The date of the delta corresponding to the "%I%" keyword.
%R%	The current release number, *i.e.*, the first component of the "%I%" keyword.
%Y%	Replaced by the value of the **t** flag (set by *admin*).

Awk — A Pattern Scanning and Processing Language

This document is based on a paper by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger of Bell Laboratories.

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

Basically, *Awk* scans a set of input lines in order, searches for lines containing any of a set of user-defined patterns, and performs a specific action on the matching lines.

Readers familiar with the UNIX† program *grep* unix program manual will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk program [files]
```

executes the *awk* commands in the string **program** on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

```
awk -f pfile [files]
```

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern   { action }
pattern   { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

†UNIX is a Trademark of Bell Laboratories.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as **$1, $2,** and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument **– F***c* may also be used to set **FS** to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, **$1**, on the file **foo1**, and the second field on file **foo2**. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process; for instance,

print | "mail bwk"

mails the output to **bwk**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

Awk also provides the **printf** statement for output formatting:

printf format, expr, expr, ...

formats the expressions in the list according to the specification in **format** and prints them. For example,

printf "%8.2f %10ld\n", $1, $2

prints **$1** as a floating point number 8 digits wide, with two after the decimal point, and **$2** as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C. [See *C Programming Language*, Prentice Hall, 1978]

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

BEGIN { FS = ":" }
... rest of program ...

Or the input lines may be counted by

END { print NR }

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

/smith/

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

blacksmithing

Awk regular expressions include the regular expression forms found in the text editor *ed* and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: **[a– zA– Z0– 9]** is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF %2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2    { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

```
     { s1 += $1; s2 += $2 }
END{ print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, – , *, /, and %(mod). The C increment ++ and decrement – – operators are also available, and so are the assignment operators +=, – =, *=, /=, and %=. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
    $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string **s** into **array**[1], ..., **array**[n]. The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the **NR**-th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
    { x[NR] = $0 }
END{ ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/  { x["apple"]++ }
/orange/{ x["orange"]++ }
END      { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like <, <=, >, >=, == ("is equal to"), and != ("not equal to"); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y     # this is a comment
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed* unix programm manual provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex lesk lexical analyzer cstr provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the operating system. The grammar is specified with *yacc*; yacc johnson cstr the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the system programs *wc, grep, egrep, fgrep, sed, lex,* and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls* – *l*; each line has the form

– rw– rw– rw– 1 ava 123 Oct 15 17:05 xxx

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc, sed,* or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk, sed* and *lex*.

Table I. Execution Times of Programs. (Times are in sec.)

	Task							
Program	1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. `END{print NR}`

2. `/doug/`

3. `/ken|doug|dmr/`

4. `{print $3}`

5. `{print $3, $2}`

6.
```
/ken/	{print >"jken"}
/doug/	{print >"jdoug"}
/dmr/	{print >"jdmr"}
```

7. `{print NR ": " $0}`

8.
```
	{sum = sum + $4}
END{print sum}
```

SED:

1. `$=`

2. `/doug/p`

3.
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4. `/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p`

5. `/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p`

6.
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1.
```
%{
int i;
%}
%%
\n	i++;
.	;
%%
yywrap() {
	printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$	printf("%s\n", yytext);
.	;
\n	;
```

DC – An Interactive Desk Calculator

This document is based on a paper by Robert Morris and Lorinda Cherry of Bell Laboratories.

DC is an arbitrary-precision arithmetic package in the form of an interactive desk calculator. It is a stack-oriented calculator using reverse Polish notation. DC ordinarily operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A– F which are treated as digits with values 10– 15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ – * % ^

The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

s*x*

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l*x*

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[...]

puts the bracketed character string onto the top of the stack.

q

exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

<*x* >*x* =*x* !<*x* !>*x* !=*x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

!

interprets the rest of the line as a system command. Control returns to DC when the system command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If **o** is capitalized, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

DETAILED DESCRIPTION

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0– 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always – 1 and all other digits are in the range 0– 99. The digit preceding the high order – 1 digit is never a 99. The representation of – 157 is 43,98,– 1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,– 1 by the digit – 1. In any case, digits which are not in the range 0– 99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned.

Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A– F correspond to the numbers 10– 15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push the value of the input base on the stack.

Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register **x**. *x* can be any character. **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in [] pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

Internal Registers – Programming DC

The load and store commands together with [] to store strings, **x** to execute and the testing commands '<', '>', '=', '!<', '!>', '!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+  si  li10>a]sa
0si  lax
```

Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as push-down stacks. **l** doesn't effect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. **;***x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command ! interprets the rest of the line as a system command and passes it to ROS to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC – An Arbitrary Precision Desk-Calculator Language.*

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

BC – An Arbitrary Precision Desk-Calculator Language

This document is based on a paper by Lorinda Cherry and Robert Morris of Bell Laboratories.

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX† time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators – , *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+– 3
```

is interpreted to mean that – 3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and – . Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c  and  (a*b)*c
```

†UNIX is a Trademark of Bell Laboratories.

BC shares with Fortran and C the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A– F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10– 15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

```
x=y=z  is the same as    x=(y=z)
x =+ y                   x = x+y
x =- y                   x = x- y
x =* y                   x = x*y
x =/ y                   x = x/y
x =% y                   x = x%y
x =^ y                   x = x^y
x++                      (x=x+1)- 1
x--                      (x=x- 1)+1
++x                      x = x+1
--x                      x = x- 1
```

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=- y and x= - y. The first replaces x by x- y and the second by - y.

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

```
bc - l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines,* Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC - An Interactive Desk Calculator.*

Appendix

1. NOTATION

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [ ] is optional.

2. TOKENS

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

2.1. Comments

Comments are introduced by the characters /* and terminated by */.

2.2. Identifiers

There are three kinds of identifiers – ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

2.3. Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A**– **F** are also recognized as digits with values 10– 15, respectively.

3. EXPRESSIONS

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

3.1. Primitive expressions

3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

3.1.1.2. *array-name* [*expression*]

Array elements are named expressions. They have an initial value of zero.

3.1.1.3. scale, ibase and obase

The internal registers **scale, ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

3.1.2. Function calls

3.1.2.1. *function-name* ([*expression* [, *expression* . . .]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

3.1.2.2. sqrt(*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale,** whichever is larger.

3.1.2.3. length(*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

3.1.2.4. scale(*expression*)

The result is the scale of the expression. The scale of the result is zero.

3.1.3. Constants

Constants are primitive expressions.

3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. – *expression*

The result is the negative of the expression.

3.2.2. ++ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

3.2.3. – – *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

3.2.4. *named-expression* ++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

3.2.5. *named-expression*– –

The named expression is decremented by one. The result is the value of the named expression before decrementing.

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

min ($a \times b$, max (**scale**, a))

3.4. Multiplicative operators

The operators *, /, % bind left to right.

3.4.1. *expression* * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

min ($a + b$, max (**scale**, a, b))

3.4.2. *expression* / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

3.4.3. *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is $a - a/b*b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

3.5. Additive operators

The additive operators bind left to right.

3.5.1. *expression* + *expression*

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.5.2. *expression* – *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.6. Assignment operators

The assignment operators bind right to left.

3.6.1. *named-expression* = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

3.6.2. *named-expression* =+ *expression*

3.6.3. *named-expression* =– *expression*

3.6.4. *named-expression* =* *expression*

3.6.5. *named-expression* =/ *expression*

3.6.6. *named-expression* =% *expression*

3.6.7. *named-expression* =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4. RELATIONS

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

4.1. *expression* < *expression*

4.2. *expression* > *expression*

4.3. *expression* <= *expression*

4.4. *expression* >= *expression*

4.5. *expression* == *expression*

4.6. *expression* != *expression*

5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

6.5. While statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

6.6. For statements

for (*expression*; *relation*; *expression*) *statement*

The for statement is the same as

```
first-expression
while(relation) {
        statement
        last-expression
}
```

All three expressions must be present.

6.7. Break statements

break

break causes termination of a **for** or **while** statement.

6.8. Auto statements

auto *identifier* [,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

6.9. Define statements

define([*parameter*[, *parameter*. . .]] **)** {
    statements }

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

6.10. Return statements

return

return(*expression* **)**

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for,** or **while** statement.

MAIL Message System - Reference Manual

This document is based on a paper by Kurt Shoens of the University of California at Berkeley.

1. INTRODUCTION

Mail lets you send and receive mail from other users on the system or network. It provides a set of *ed*-like commands for manipulating messages and sending mail.

This document describes Mail for both casual and frequent users of the program. The reader is assumed to be familiar with the UNIX[1] Shell, the text editor, and some of the common UNIX commands. If you are a neophyte Mail user, section two of this document should provide enough information to allow you to effectively use Mail. The balance of this document describes more advanced features, which are useful to those commonly barraged with a large volume of mail.

2. COMMON USAGE

The Mail command has two distinct usages, according to whether one wants to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, "root," use the Shell command:

```
%Mail root
```

then type your message. When you reach the end of the message, type an EOT (control– d) at the beginning of a line, which will cause Mail to echo "EOT" and return you to the Shell. When the user you sent mail to next logs in, he will receive the message:

```
You have mail.
```

to alert him to the existence of your message. Incidentally, once you have sent mail to someone, there is no way to undo the act, so be careful. The message your recipient reads will consist of the message you typed, preceded by a line telling who sent the message (your login name), the teletype from which the message was sent, and the date and time it was sent.

If you want to send the same message to several other people, you can list all of their login names on the command line. Thus,

```
%Mail sam bob john
Tuition fees are due next Friday. Don't forget!!
<Control– d>
EOT
%
```

will send the reminder to sam, bob, and john.

If, when you log in, you see the message,

```
You have mail.
```

you can read the mail by typing simply:

```
%Mail
```

Mail will respond by typing its version number and date and then listing the messages you have waiting. Then it will type an underscore and await your command. The messages are assigned numbers starting with 1 – you can refer to the messages with these numbers.

[1] UNIX is a trademark of Bell Laboratories.

To look at a specific message, use the **type** command, which may be abbreviated to simply **t**. For example, if you had the following messages:

```
1 root     Wed Sep 21 09:21  "Tuition fees"
2 sam      Tue Sep 20 22:55
```

you could examine the first message by giving the command:

```
type 1
```

which might cause Mail to respond with, for example:

```
Message  1:
From root  tty8  Wed Sep 21 09:21:45 1978
Subj: Tuition fees

Tuition fees are due next Wednesday.  Don't forget!!
```

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave Mail. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox* you can delete it using the **delete** command. In our example,

```
delete 1
```

will prevent Mail from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, Mail will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The **delete** command can be abbreviated to simply **d**.

When you have perused all of the messages of interest, you can leave Mail with the **quit** command, which saves all of the messages you have typed but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

```
%Mail
```

The **quit** command can be abbreviated to simply **q**.

If you wish for some reason to leave Mail quickly without altering either your system mailbox or *mbox*, you can type the **x** command (short for **exit**), which will immediately return you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving Mail, you can type the command preceded by an exclamation point, just as in the text editor. Thus, for instance:

```
!date
```

will print the current date without leaving Mail.

Finally, the **help** command is available to print out a brief summary of the Mail commands, using only the single character command abbreviations.

3. TILDE ESCAPES

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or perform some other auxiliary function. Mail provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

```
~p
```

which will print a line of dashes, the recipients of your message, and the text of the message so far. If you are dissatisfied with the message as it stands, you can invoke the text editor on it using the escape

```
~e
```

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. Mail will respond by typing

    (continue)

after which you may continue typing text which will be appended to your message, or type <control-d> to end the message.

It is often useful to be able to include the contents of some file in your message; the escape

    ~r filename

is provided for this purpose, and causes the named file to be appended to your current message. Mail complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text.

As a special case of ~r, the escape

    ~d

reads in the file "dead.letter" in your home directory. This is often useful since Mail copies the text of your message there when you abort a message with RUBOUT.

In order to save the current text of your message on a file you may use the

    ~w filename

escape. Mail will print out the number of lines and characters written to the file, after which you may continue appending text to your message.

If you are sending mail from within Mail's command mode (read about the **reply** and **mail** commands, section six), you can read a message sent to you into the message you are constructing with the escape:

    ~m 4

which will read message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

    ~t name1 name2 ...

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; in fact, you cannot remove someone from the recipient list with ~t.

If you wish, you can associate a subject with your message by using the escape

    ~s Arbitrary string of text

which replaces any previous subject with "Arbitrary string of text." The subject, if given, is sent near the top of the message prefixed with "Subj:" You can see what the message will look like by using ~p.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

    ~c name1 name2 ...

adds the named people to the "Cc:" list, similar to ~t. Again, you can execute ~p to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subj:" field, and the carbon copies the "Cc:" field. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape

```
~h
```

which prints "To:" followed by the current list of recipients and leaves the cursor (or print-head) at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard # and @ symbols,

```
~h
To: root kurt####bill
```

would change the initial recipients "root kurt" to "root bill." When you type a newline, Mail advances to the "Subj:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use ~p to print the current text of the header fields and the body of the message.

To effect a temporary escape to the shell, the escape

```
~!command
```

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

```
~|command
```

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, Mail assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt* which is designed to format outgoing mail.

If you wish (for some reason) to send a message which contains a line beginning with a tilde, you must double it. Thus, for example,

```
~~This line begins with a tilde.
```

sends the line

```
~This line begins with a tilde.
```

Finally, the escape

```
~?
```

prints out a brief summary of the available tilde escapes.

4. MESSAGE LISTS

The **type** and **delete** commands described in section two take a list of messages as argument, as do many of the commands described in section six. This section describes the construction of message lists in general.

A *message list* consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters "↑" "." or "$" to specify the first relevant, current, or last relevant message, respectively. *Relevant* here means, for most commands "not deleted" and "deleted" for the **undelete** command.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

```
type 1– 4
```

and to print all the messages from the current message to the last message, use

```
type .– $
```

A *name* is a user name. All of the user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users which is *relevant* (in the sense described earlier) is selected. Thus, to print every message sent to you by "root," do

```
type root
```

As a shorthand notation, you can specify simply "*" to get every *relevant* (same sense) message. Thus,

```
type *
```

prints all undeleted messages,

```
delete *
```

deletes all undeleted messages, and

```
undelete *
```

undeletes all deleted messages.

5. COMMAND LINE OPTIONS

This section describes the alternate usages of Mail from the shell.

As you continue to receive system mail, you will most likely accumulate a large collection of messages in the file *mbox*. In order to help you deal with this, Mail allows you to edit files of messages by using the **-f** flag. Specifically,

```
Mail – f filename
```

causes Mail to edit "filename" and

```
Mail – f
```

causes Mail to read "mbox" in your home directory. All of the Mail commands except **preserve** are available to edit the messages. When you type the **quit** command, Mail will write the updated file back.

Since you will usually have a large number of messages stored in *mbox*, Mail will only print out the first 18 message headers when editing more than 18 messages. To display the other message headers, the **headers** command takes as an optional argument either + or – to move forward or back to the next or previous 18 message group.

If you send mail over a noisy phone line, you will notice that many of the garbage characters turn out to be the RUBOUT character, which causes Mail to abort the message. To deal with this annoyance, you can invoke Mail with the **-i** option to causes these garbage characters to be ignored. Unfortunately, as you are typing in a line of text to a program, the little gnome which gathers up the characters is instructed to throw them all away when a RUBOUT is seen. For this reason, Mail indicates that a RUBOUT has been received by echoing an @ to tell you that everything you had typed on that line has been thrown away.

6. ADDITIONAL COMMANDS

This section describes additional Mail commands available when receiving mail.

The **next** command goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

```
type root
```

goes to the next message sent by "root" and types it. The **next** command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters "↑" "." or "$". Thus, prints the current message and

```
    4
```

prints message 4.

The – command goes to the previous message and prints it. The – command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

The **save** command allows you to save messages received from others on a file other than *mbox*. Its syntax varies somewhat from the other commands which accept a message list in that the final word on the command line is taken to be the file on which to save the messages. The named messages are appended to the file (which is created if it did not already exist) and are marked as saved. Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, unless explicitly specified. The **save** command provides a facility for saving messages pertaining to a particular subject or from a particular person in a special place.

The **undelete** command causes a message which had been deleted previously to regain its initial status. Only messages which are already deleted may be undeleted. This command may be abbreviated to **u**.

The **preserve** command takes a message list and marks each message therein so that it will be saved in your system mailbox instead of being deleted or saved in *mbox* when you quit. This is useful for saving messages of importance that you want to see again, or messages not intended for you if you are sharing a login name.

Often, one wants to deal with a message by responding to its author right then and there. The **reply** command is useful for this purpose: it takes a message list and sends mail to the authors of those messages. The message is collected in the usual fashion by reading up to an EOT. All of the tilde escapes described in section three will work in **reply**. Additionally, if there are header fields in the message being replied to, this information is copied into the new message. The **reply** command can be abbreviated to **r**.

In order to simply mail to a user inside of Mail, the **mail** command is provided. This sends mail in the manner described for the **reply** command above, except that the user supplies a list of recipient login names and distribution groups. All of the tilde escapes described in section three will work in **mail**. The **mail** command may be abbreviated to **m**.

In order to edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of message as described under the **type** command and processes each by writing it into the file Message*x* where *x* is the message number being edited and executing the text editor on it. When you have edited the message to your satisfaction, write the message out and quit, upon which Mail will read the message back and remove the file. **Edit** may be abbreviated to **e**.

It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

When Mail is invoked to receive mail, it prints out the message header for each message. In order to reprint the headers for remaining messages (those which haven't been deleted), you may type the **headers** command. Deleted messages do not appear in the listing, saved messages are flagged with a "*" and preserved messages are flagged with a "P."

The **from** command takes a list of messages and prints out the header lines for each one; hence

```
    from joe
```

is the easy way to display all the message headers from "joe."

The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**.

The **dt** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail.

7. SUMMARY OF COMMANDS, ESCAPES, AND OPTIONS

This sections describes tersely all of the Mail commands, escapes, and options. For each command, its most abbreviated form (in brackets) and a short description of the command is given below.

First, message lists are computed by determining the set M which consists of all message referenced explicitly or through ranges. Then, the set U is computed, which consists of all messages sent by *any* of the user names specified. Finally, the message list is calculated by finding the intersection of sets M and U.

Each Mail command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety – the first command which matches the typed prefix is used. If the argument begins with a digit or special character, then no space is required following the command letter, but otherwise the space is required. If a Mail command does not take arguments, they may be specified, even though they are ignored. For the commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, Mail types "No applicable messages" and aborts the command.

–	[–] Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n*th previous message and prints it. If there is no previous message, it prints "Nonzero address required."
=	[=] Prints out the current message number. Takes no arguments.
?	[?] Prints out the file /usr/lib/Mail.help, which contains a brief summary of the commands. Takes no arguments.
!	[!] Executes the UNIX Shell command which follows. Unlike other commands, there does not need to be a space after the exclamation point.
alias	[a] With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, adds the users named in the second and later arguments to the alias named in the first argument.
chdir	[c] Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.
delete	[d] Takes a list of messages as argument and marks them all as deleted. Deleted messages will not be saved in *mbox*, nor will they be available for most other commands. Default messages may not be deleted already.
dp	[dp] Deletes the current message and prints the next message. If there is no next message, types out "At EOF."
dt	[dt] Same as **dp**.
edit	[e] Takes a list of messages and points the text editor at each one in turn. On return from the editor, the message is read back in. The default message for **edit** may not be saved or deleted.
exit	[ex] Effects an immediate return to the Shell without modifying the user's system mailbox, his *mbox* file, or his edit file in – **f**.
from	[f] Takes a list of messages and prints their message headers. The default message is neither saved nor deleted.
headers	[h] Lists the current range of headers, which is an 18 message group. If the "+" argument is given, then the next 18 message group is printed, and if the "–" argument is given, the previous 18 message group is printed.

help	[hel] A synonym for ?
hold	[ho] Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command. The default message must not be deleted.
list	[l] The **list** command lists all of the available user commands in the order that the command processor sees them. It takes no arguments.
mail	[m] Takes as argument login names and distribution group names and sends mail to those people. Tilde escapes work in **mail.**
next	[n] Goes to the next message in sequence and types it. If a message list is given, then **next** goes to the first message in the message list.
preserve	[pre] A synonym for **hold.**
print	[p] Takes a message list and types out each message on the user's terminal. The default message must not be deleted.
quit	[q] Terminates the Mail session, saving all undeleted, unsaved messages in the user's *mbox* file in his login directory, preserving all messages marked with **hold** or **preserve** in his system mailbox, and removing all other messages from his system mailbox. If mail has arrived during the Mail session, the message "You have new mail" is typed. If executing while editing a mailbox file with the – **f** flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.
reply	[r] Takes a message list and sends mail to each message author just like the **mail** command. The default message must not be deleted.
respond	[r] A synonym for **reply**.
save	[s] Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal. The default message for **save** may not be saved or deleted.
set	[se] With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."
shell	[sh] Invokes an interactive version of the shell.
size	[si] Takes a message list and prints out the size in characters of each message. The default message for **size** must not be deleted.
top	[to] Takes a message list and prints the top so many lines. The number of lines printed is controlled by the variable "toplines" and defaults to five.
type	[t] A synonym for **print.**
unalias	[una] Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.
undelete	[u] Takes a message list and marks each one as *not* being deleted. Each message in the list must already be deleted. The default message must be deleted.
unset	[uns] Takes a list of option names and discards their remembered values; opposite of **set.**
visual	[v] Takes a message list and invokes the display editor on each one.
write	[w] A synonym for **save.**
xit	[x] A synonym for **exit.**

Recall that tilde escapes are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option "escape."

Here is a summary of the tilde escapes:

~!command	Execute the indicated shell command, then return to the message.
~c name ...	Add the given names to the list of carbon copy recipients.
~d	Read the file "dead.letter" from your home directory into the message.
~e	Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.
~h	Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.
~m messages	Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.
~p	Print out the message collected so far, prefaced by the message header fields.
~q	Abort the message being sent, copying the message to "dead.letter" in your home directory if "save" is set.
~r filename	Read the named file into the message.
~s string	Cause the named string to become the current subject field.
~t name ...	Add the given names to the direct recipient list.
~v	Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a visual editor. After you quit the editor, you may resume appending text to the end of your message.
~w filename	Write the message onto the named file.
~|command	Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of the message.
~~string	Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double *that* character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case it's actual value is of interest.

The binary options include the following:

append	Causes messages saved in *mbox* to be appended to the end rather than prepended.
ask	Causes Mail to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.
askcc	Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.
autoprint	Causes the **delete** command to behave like **dp** – thus, after deleting a message, the next one will be typed automatically.
ignore	Causes interrupt signals from your terminal to be ignored and echoed as @'s.
metoo	Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.
quiet	Suppresses the printing of the version when Mail is first invoked.
save	Causes the message collected prior to a RUBOUT to be saved on the file "dead.letter" in your home directory on receipt of the RUBOUT. Also causes

the message to be so saved in the same fashion for ~q.

The following options have string values:

EDITOR	Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.
SHELL	Pathname of the shell to use in the **!** command and the ~! escape. A default shell is used if this option is not defined.
VISUAL	Pathname of the text editor to use in the **visual** command and ~v escape.
escape	If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.
record	If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not so saved.
toplines	If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

8. CONCLUSION

I would like to acknowledge the help of Eric Allman, Ken Arnold, Bob Fabry, Richard Fateman, Bob Kridle, Doug Merritt, David Mosher, Eric Schmidt, Polly Siegel, Michael Ubell, and Bill Joy.

M4 – a Macro Processor

This document is based on a paper by Brian W. Kernighan and Dennis M. Ritchie of Bell Laboratories.

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is '– ', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines N to be 100, and uses this ``symbolic constant'' in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of N's.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In M4, the latter is true — M is 100, so even if N subsequently changes, M does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string N is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when you ask for M later, you'll always get the value of N at that time (because the M will be replaced by N which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N´)
```

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
        `define´ = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining N:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the N in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of N. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)')
ifdef(`gcos', `define(wordsize,36)')
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix', on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

define(bump, $1 = $1 + 1)

generates code to increment its argument by 1:

bump(x)

is

x = x + 1

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

define(cat, $1$2$3$4$5$6$7$8$9)

Thus

cat(x, y, z)

is equivalent to

xyz

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

define(a,   b   c)

defines **a** to be **b   c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

define(a, (b,c))

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

define(N, 100)
define(N1, `incr(N)')

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^      (exponentiation)
* / %  (modulus)
+ -
== != < <= > >=
!            (not)
& or &&      (logical and)
| or ||      (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2**N+1**. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (``silent include'') says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(a, b, c, d)

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns ``yes'' or ``no'' if they are the same or different.

define(compare, `ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr(`now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or − 1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
      define(...)
      ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

```
errprint(`fatal error')
```

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

3	changequote(L, R)
1	define(name, replacement)
4	divert(number)
4	divnum
5	dnl
5	dumpdef(`name', `name', ...)
5	errprint(s, s, ...)
4	eval(numeric expression)
3	ifdef(`name', this if true, this if false)
5	ifelse(a, b, c, d)
4	include(file)
3	incr(number)
5	index(s1, s2)
5	len(string)
4	maketemp(...XXXXX...)
4	sinclude(file)
5	substr(string, position, number)
4	syscmd(s)
5	translit(str, from, to)
3	undefine(`name')
4	undivert(number,number,...)

Acknowledgements

Thanks to Rick Becker, John Chambers, Doug McIlroy, and Jim Weythman.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Inc., 1976.

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956