IBM

ACADEMIC OPERATING SYSTEM 4.3

ACADEMIC OPERATING SYSTEM

VOLUME III

IBM

ACADEMIC OPERATING SYSTEM 4.3

ACADEMIC OPERATING SYSTEM

VOLUME III

IBM Academic Operating System 4.3

Volume III

Assembler Reference Manual for IBM/4.3

ABSTRACT

This article is an updated version of an article entitled *Berkeley VAX/UNIX Assembler Reference Manual,* written in November 1979 by John F. Reiser and Robert R. Henry and revised in February 1983. The original article, which is in Volume 1 of *UNIX Programmer's Supplementary Documents*, has been rewritten and includes additions and changes for IBM/4.3 and corrections where appropriate.

1. INTRODUCTION

This document describes the usage and input syntax of the IBM/4.3 assembler, *as*, for the IBM RT PC and IBM 6152 Academic System. *As* assembles the code produced by the C compiler. This article is intended for those writing a compiler or maintaining the assembler; it is not a user's guide for writing assembler code.

Examples of syntax in this article use the following conventions:

- [Argument] means that the specified argument is optional; 0 or more instances may be included.
- Words in **boldface** must appear literally.
- Words in *italics* represent specific values to be supplied.

2. USAGE

As is invoked with these command arguments:

as [**−LVWRDT**] [**−t** *directory*] [**−o** *outfile*] [$name_1$] . . . [$name_n$]

The arguments are explained below:

-L Instructs the assembler to save labels beginning with an "L" in the symbol table portion of the file specified as *outfile*. Labels are not saved by default, as the default action of the link editor *ld* is to discard them anyway.

-V Tells the assembler to place its interpass temporary file in virtual memory. In normal circumstances, the system manager will decide where the temporary file should lie. Experiments with a temporary file of 115 kbytes have shown this option to have a negligible (1-2%) effect on assembly time on an unloaded machine.

-W Turns off all warning error reporting.

-R Make initialized data segments read-only by concatenating them to the text segments. This obviates the need to run editor scripts on assembler source to "read−only" fix initialized data segments. Uninitialized data (via **.lcomm** and **.comm** directives) are still assembled into the bss segment.

-D Prints assembler debugging information and dumps the symbol table, provided the assembler has been compiled with DEBUG defined.

-T Prints the token file, provided the assembler has been compiled with DEBUG defined. This information is useful when debugging the assembler.

-t Causes the assembler to place its single temporary file in *directory* instead of in /tmp , provided the **−V** flag is not set.

-o Causes the output to be placed in the file *outfile*. By default, the output of the assembler is placed in the file a.out in the current directory.

$name_{1\text{-}n}$ Causes input to be taken sequentially from the files $name_1$ . . . $name_n$. The files are not assembled separately; $name_1$ is effectively concatenated to $name_2$ so multiple definitions cannot occur among the input sources. By default, input is taken from the standard input.

Note: Arguments **-J** and **-d** are ignored.

3. LEXICAL CONVENTIONS

Assembler tokens include identifiers (alternatively, "symbols" or "names"), constants, and operators.

3.1. Identifiers

An identifier consists of a sequence of alphanumeric characters, including the special characters period (.), underscore (_), and dollar ($). The first character may not be a digit or a dollar sign. For all practical purposes, the length of identifiers is arbitrary; all characters are significant. All keywords, operation mnemonics, register names, and macro names are reserved and are not available as user-defined names.

3.2. Constants

3.2.1. Integral Constants

All integral (non floating point) constants are (potentially) 64 bits wide. Integral constants are initially evaluated to a full 64 bits, but are pared down by discarding high order copies of the sign bit and categorizing the number as a long (32 bits) or double-long (64 bits) integer. Numbers with less precision than 32 bits are treated as 32-bit quantities. *As* cannot perform arithmetic on constants larger than 32 bits and supports 64-bit integers only so they can be used to fill initialized data space.

The digits are "0123456789abcdefABCDEF" with the obvious values.

A decimal constant consists of a sequence of digits without a leading zero.

An octal constant consists of a sequence of digits with a leading zero.

A hexadecimal constant consists of the characters "0x" (or "0X") followed by a sequence of digits.

A single-character constant consists of a single quote (') followed by an ASCII character, including ASCII newline. The constant's value is the code for the given character.

3.2.2. Floating Point Constants

IEEE single and double precision constants are supported by the **.float** and **.double** directives respectively. The *atof*(3) man page describes the range of representable values and their syntax. There is presently no support for IEEE double extended precision constants. For a description of the IEEE representations, please see the *IEEE Standard 754 for Binary Floating Point Arithmetic.* The assembler uses the library routine *atof*(3) to convert floating point numbers.

The operand field syntax of .float and .double is:

$$0[expe]([+-])\ [dec]^{+}(.)([dec]^{*})([expt]([+-])([dec]^{+}))$$

where:

expe	An exponent delimiter and type specification character (fFdD).
dec	A decimal digit (0 1 2 3 4 5 6 7 8 9).
expt	A type specification character (eEfFdD).
x^{*}	0 or more occurrences of x.
x^{+}	1 or more occurrences of x.

The standard semantic interpretation is used for the signed integer, fraction and signed power of 10 exponent. If the exponent delimiter is specified, it must be either an "e" or "E", or must agree with the initial type specification character that is used. A .double constant must have d or D specified as its type specification character; a .float constant must have f or F specified as its type specification character.

Collectively, all floating point numbers, together with double-long integral numbers, are called "bignums". When *as* requires a bignum, a 32-bit scalar quantity may also be used.

3.2.3. String Constants

A string constant is defined using the same syntax and semantics as the C language uses. Strings begin and end with a double quote ("). All C backslash conventions are observed. Strings are known by their value and their length; the assembler does not implicitly end strings with a null byte.

3.3. Operators

There are several single-character operators; see Section 6.1.

3.4. Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5. Single Line Comments

The character "#" introduces a comment which extends through the end of the line. Comments starting in column 1, having the format *"# expression string"*, are interpreted as an indication that the assembler is now assembling file *string* at line *expression*. Thus, one can use the C preprocessor on an assembly language source file, and use the *#include* and *#define* preprocessor directives. Other comments may not start in column 1 if the assembler source is given to the C preprocessor because the preprocessor will misinterpret them. Comments are otherwise ignored by the assembler.

To retain compatibility with existing .s files, comments beginning with "|" are also accepted. However, this use is deprecated, and support for this feature will be removed in subsequent releases.

3.6. C Style Comments

The assembler will recognize C style comments, introduced with the prologue /* and ending with the epilogue */. C style comments may extend across multiple lines and are the preferred comment style to use if you choose to use the C preprocessor.

If a C style comment does extend across "n" lines, the line numbers in any subsequent error messages generated by the assembler will be low by n-1 lines, since the assembler increments the line count only once for a multiple C style comment.

4. SEGMENTS AND LOCATION COUNTERS

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The operating system makes some assumptions about the content of these segments; the assembler does not. Within the text and data segments there are a number of sub-segments, distinguished by number ("**text** 0", "**text** 1", "**data** 0", "**data** 1" , . . .). Currently there are four subsegments each in text and data. The subsegments are for programming convenience only.

Before writing the output file, the assembler zero-pads each text subsegment to a multiple of eight bytes and then concatenates the subsegments in order to form the text segment; an analogous operation is done for the data segment. Requesting that the loader define symbols and storage regions is the only action allowed by the assembler with respect to the bss segment. Assembly begins in "**text** 0".

Associated with each (sub)segment is an implicit location counter which begins at zero and is incremented by 1 for each byte assembled into the (sub)segment. There is no way to explicitly reference a location counter. Note that the location counters of subsegments other than "**text** 0" and "**data** 0" behave peculiarly due to the concatenation used to form the text and data segments.

5. STATEMENTS

A source program is composed of a sequence of statements. Statements are separated by newlines or by semicolons. There are two kinds of statements: null statements and keyword statements. Either kind of statement may be preceded by one or more labels.

5.1. Named Labels

A named label consists of a name followed by a colon. The effect of a named label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

Named labels beginning with an "L" are not retained in the a.out symbol table unless the **-L** option is in effect.

5.2. Numeric Local Labels

A numeric label consists of a digit between 0 and 9 followed by a colon. A numeric label defines temporary symbols of the form "*n*b" and "*n*f" where *n* is the digit of the label. As in the case of named labels, a numeric label assigns the current value and type of the location counter to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References to symbols of the form "*n*b" refer to the first numeric label *n: b*ackward from the reference; "*n*f" symbols refer to the first numeric label *n: f*orward from the reference.

As turns local labels into labels of the form L*n*\001m for internal purposes.

5.3. Null Statements

A null statement is an empty statement ignored by the assembler. A null statement may be labeled, however.

5.4. Keyword Statements

A keyword statement begins with one of the many predefined keywords known to *as*; the syntax of the remainder of the statement depends on the keyword. All instruction opcodes, listed in Section 8, are keywords. The remaining keywords are assembler pseudo-operations, also called "directives." The pseudo-operations are listed in Section 7, together with the syntax they require.

6. EXPRESSIONS

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, operators, and parentheses. Each expression has a type.

All operators in expressions are fundamentally binary in nature. Arithmetic is two's complement and has 32 bits of precision. *As* cannot perform arithmetic operations on floating point numbers or on double-long integral numbers. There are four levels of precedence, listed here from lowest precedence level to highest:

precedence	operators
binary	+ −
binary	& ^ !
binary	* / %
unary	− ~

All operators of the same precedence are evaluated strictly left to right, except for the evaluation order enforced by parentheses.

6.1. Expression Operators

The operators are:

operator	meaning
+	addition
−	(binary) subtraction
*	multiplication
/	division
%	modulo
−	(unary) two's complement
&	bitwise and
^	bitwise exclusive or
!	bitwise or not
~	bitwise ones' complement
>	logical right shift
> >	logical right shift
<	logical left shift
< <	logical left shift

Expressions may be grouped with parentheses.

6.2. Data Types

Every user-defined symbol has one of the following types. The type propagation rules in the next section describe how expression types are derived from symbol types.

undefined Upon first encounter, each symbol is undefined unless its first encounter defines it. It may become undefined if it is assigned an undefined expression. The assembler changes all undefined types to undefined external just prior to pass 2.

undefined external
A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined in a **.set** by an expression of type absolute. Constants have type absolute.

text A symbol appearing as a label in a text segment has type text, as does a symbol defined in a **.set** by an expression of type text. The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output.

data A symbol appearing as a label in a data segment has type data, as does a symbol defined in a **.set** by an expression of type data. The value of a data symbol is measured with respect to the origin of the data segment of a program. The value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments.

bss A symbol defined in a **.comm** or **.lcomm** directive has type bss, as does a symbol defined in a **.set** by an expression of type bss. The value of a bss symbol is measured from the beginning of the bss segment of a program. The value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments.

external absolute, text, data, or bss
Symbols declared **.globl** and defined within an assembly as absolute, text, data, or bss types may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

6.3. Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation, the important types are:

undefined
absolute
text
data
bss
undefined external
relocatable: any of text, data, bss, or undefined external

The combination rules are:

(1) If one of the operands is undefined, the result is undefined.

(2) If both operands are absolute, the result is absolute.

(3) An absolute operand may be added to or subtracted from any other type, and the type of the result is that of the other operand.

(4) An operand of type text, data, or bss may be subtracted from an operand having the same type, and the type of the result is absolute.

(5) Any other combination is an error.

7. PSEUDO-OPERATIONS (DIRECTIVES)

The keywords listed below introduce pseudo-operations (directives) to influence the later behavior of the assembler, define symbols, or create data. They are grouped below into functional categories.

7.1. Interface to a Previous Pass

.ABORT

As soon as the assembler sees this directive, it ignores all further input (but it does read to the end of file) and aborts the assembly. No files are created. It is anticipated that this would be used in a pipe interconnected version of a compiler, where the first major syntax error would cause the compiler to issue this directive, saving unnecessary work in assembling code that would have to be discarded anyway.

.file *string*

This directive causes the assembler to think it is in file *string*, so that error messages reflect the proper source file.

.line *expression*

This directive causes the assembler to think it is on line *expression* so that error messages reflect the proper source line.

The only effect of assembling multiple files specified in the command string is to insert the *file* and *line* directives, with the appropriate values, at the beginning of the source from each file.

expression string

This is the only instance where a comment is meaningful to the assembler. The "#" must be in the first column. This meta comment causes the assembler to believe it is on line *expression*. The second argument, if included, causes the assembler to believe it is in file *string*; otherwise the current file name does not change.

7.2. Location Counter Control

.data [*expression*]
.text [*expression*]

These two directives cause the assembler to begin assembling into the indicated text or data subsegment. If specified, *expression* must be defined and absolute; an omitted expression is treated as zero. Assembly starts in the **.text** 0 subsegment.

The directives **.align** and **.org** also control the placement of the location counter.

While the comments within the assembler may refer to the location counter as "." or "dot", there is no explicit reference allowed to the location counter. Numeric local labels may be used with almost equal convenience and more predictable results.

7.3. Filled Data

.align *align_expr*

The location counter is adjusted so that the *align_expr* lowest bits of the location counter become zero. This is done by assembling from 0 to 2^{align_expr} -1 bytes of 0. Thus ".align 2" pads by null bytes to make the location counter evenly divisible by 4. The *align_expr* must be defined, absolute, nonnegative, and less than 16.

Warning: the subsegment concatenation convention and the current loader conventions may not preserve attempts at aligning to more than 3 low-order zero bits.

.org *org_expr*[,*fill_expr*]

The location counter is set equal to the value of *org_expr*, which must be of type text or data and greater than the current value of that segment's location counter. Space between the current value of the location counter and the desired value are filled with bytes taken from the low order byte of *fill_expr*, which must be absolute and defaults to 0.

.space *space_expr*[,*fill_expr*]

The location counter is advanced by *space_expr* bytes. *Space_expr* must be defined and absolute. The space is filled in with bytes taken from the low order byte of *fill_expr*, which must be defined and absolute. *Fill_expr* defaults to 0. The **.fill** directive is a more general way to accomplish the **.space** directive.

.fill *rep_expr, size_expr, fill_expr*

All three expressions must be absolute. *Fill_expr*, treated as an expression of size *size_expr* bytes, is assembled and replicated *rep_expr* times. The effect is to advance the current location counter *rep_expr* * *size_expr* bytes. *Size_expr* must be between 1 and 8.

7.4. Initialized Data

.byte *expr*[,*expr*]. . .
.short *expr*[,*expr*]. . .
.int *expr*[,*expr*]. . .
.long *expr*[,*expr*]. . .

Expr represents an expression. Expressions are truncated to the size indicated by the keyword in the table below, and assembled in successive locations. Non-absolute expressions in a **.byte** or **.short** engender a warning message.

keyword	length (bits)
.byte	8
.short	16
.int	32
.long	32

Each expression may optionally be of the form:

$expression_1$: $expression_2$

In this case, the value of $expression_2$ is truncated to $expression_1$ bits, and assembled in the next $expression_1$ bit field which fits in the natural data size being assembled. Bits which are skipped because a field does not fit are filled with zeros. Thus, "**.byte** 123" is equivalent to "**.byte** 8:123", and "**.byte** 3:1,2:1,5:1" assembles two bytes, containing the values 0x28 and 0x08.

.dlong *number*[,*number*]. . .
.float *number*[,*number*]. . .
.double *number*[,*number*]. . .

These initialize bignums (see Section 3.2.2) in successive locations whose size is a function of the keyword. The type of the bignum (determined by the exponent field, or lack thereof) may not agree with the type implied by the keyword. The following table shows the keywords, their size, and the data types for the bignums they expect.

keyword	format	length (bits)	valid *number* (s)
.dlong	integral	64	integral
.float	ieee single	32	floating and integral
.double	ieee double	64	floating and integral

.ascii *string*[, *string*]. . .
.asciz *string*[, *string*]. . .

Each *string* in the list is assembled into successive locations, with the first letter in the string being placed into the first location, etc. The **.ascii** directive will not null terminate the string; the **.asciz** directive will null terminate the string. (Recall that strings are known by their length and need not be terminated with a null, and that the C conventions for escaping are understood.) The **.ascii** directive is identical to:

.byte $string_0$, $string_1$, . . .

.comm *name, expression*

Provided the *name* is not defined elsewhere, its type is made "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link editor *ld* has been special-cased so that all undefined external symbols that have a non-zero value are defined to lie in the bss segment, and space is reserved after the symbol to hold *expression* bytes.

.lcomm *name, expression*

Expression bytes will be allocated in the bss segment and *name* assigned the location of the first byte, but the *name* is not declared as global and hence will be unknown to the link editor.

.globl *name*

This directive makes *name* external. If it is otherwise defined (by **.set** or by appearance as a label) it acts within the assembly exactly as if the **.globl** directive were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbol.

.set *name, expression*

The (*name, expression*) pair is entered into the symbol table. Multiple **.set** statements with the same name are legal; the most recent value replaces all previous values.

.lsym *name, expression*

A unique instance of the (*name, expression*) pair is created in the symbol table. This mechanism can be used to pass local symbol definitions to the link editor and debugger. Note that *name* may not be referenced.

.stabs *string,* $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabn $expr_1$, $expr_2$, $expr_3$, exr_4
.stabd $expr_1$, $expr_2$, $expr_3$

The **.stab***x* directives place symbols in the symbol table for the symbolic debugger, *dbx*. A "stab" is a *s*ymbol *tab*le entry. The **.stabs** is a string stab, the **.stabn** is a stab not having a string, and the **.stabd** is a "dot" stab that implicitly references "dot", the current location counter.

The *string* in the **.stabs** directive is the name of a symbol. If the symbol name is zero, the **.stabn** directive may be used instead.

The other expressions are stored in the name list structure of the symbol table and preserved by the loader for reference by *dbx*; the values of the expressions are peculiar to formats required by *dbx*.

expr$_1$ Is used as a symbol table tag (nlist field *n_type*).

expr$_2$ Is always zero (nlist field *n_other*).

expr$_3$ Is used for either the source line number, or for a nesting level (nlist field *n_desc*).

expr$_4$ Is used as tag specific information (nlist field *n_value*). In the case of the **.stabd** directive, this expression is nonexistent, and is taken to be the value of the location counter at the following instruction. Since there is no associated name for a **.stabd** directive, it can be used only in circumstances where the name is zero. The effect of a **.stabd** directive can be achieved by one of the other **.stab**x directives in the following manner:

> **.stabn** $expr_1$, $expr_2$, $expr_3$, LL_n
> LL*n*:

The **.stabd** directive is preferred, because it does not clog the symbol table with labels used only for the stab symbol entries.

7.5. Addressability

.using *expr,register,. . .*

The **.using** directive tells the assembler that it can rely on the value in a register for the purpose of creating base + displacement addresses for machine instructions.

Expr may be any relocatable expression of type text or data. The register is assumed to contain an address pointing to the storage location described by the relocatable expression. Each additional specified register is assumed to contain an address 0x8000 bytes greater than the previous register.

There may be one **.using** specified for each text subsegment and one for each data subsegment (i.e. up to eight **.using**'s may be in effect at any time). If a **.using** is not provided for a .text or for a .data subsegment but is provided for a lower-numbered text or data subsegment, the one for the lower-numbered subsegment will be used. If no **.using** is provided for any text subsegment, reference to an address of type text encodes a warning message and register 11 is assumed to point to the beginning of the text 0 subsegment. If no **.using** is provided for any data subsegment, reference to an address of type data engenders an error message. If a proper register and displacement cannot be formed from a **.using** statement, an error message is issued.

If a second **.using** is specified while one is active within the same subsegment, the second replaces the first. A **.using** followed by a relocatable expression without a register unassigns the base register.

Symbols in the relocatable expression need not be defined before the appearance of the **.using** directive.

7.6. Literal Operands

The following construct may be used in machine instructions wherever a relocatable instruction operand may be used:

$.*data-directive expression*

The arguments are explained below:

data-directive	Any of **.byte**, **.short**, **.int**, **.long**, **.dlong**, **.float**, **.double**, **.ascii**, or **.asciz.**
expression	Any single expression that is legal for the respective assembler directive.

The following lines show examples of literals:

```
        lc      r1,$.byte 0x18
        lh      r2,$.short (4 < < 8)
        l       r2,$.int 123456
```

The line:

```
        l       r7,$.long root
```

is equivalent to:

```
        l       r7,Z00001
        . . .
Z00001: .long   root
```

Literals are accumulated into a pool and duplicates are removed. Literals are considered duplicates when they are written in exactly the same way; constants which assemble to the same value but which have different source forms are different literals, except that .long and .int are considered to be equal. String literals are never considered to be equal. The literal pool is sorted such that the items with the more restrictive alignment are placed first. The beginning of the literal pool is aligned to the boundary implied by the first literal in the pool.

.ltorg

This directive indicates the start of a literal pool and causes the accumulated literal values to be emitted. The **.ltorg** directive can appear in either a text or data segment, and it can appear more than once. If literals are used and no **.ltorg** follows, a warning will be issued and the literals will be emitted at the end of the .text 0 subsegment.

8. MACHINE INSTRUCTIONS

This section describes the machine instructions, extended branch mnemonics, and macro instructions supported by *as*.

8.1. Summary of Machine Instructions

The symbols used to describe the source syntax are:

abs An absolute expression representing a displacement from a base.

f An absolute value representing a register bit position.

i An absolute expression representing an immediate value, optionally preceded by a "$".

lbl A name of type text, data, or undefined external.

ra,rb,rc Register expressions. A register expression is one of the predefined symbols r0, . . . r15, sp, or a "%" followed by an absolute in the range 0-15. sp is equivalent to r1.

reloc An address operand of one of the following forms:
abs(register-expression)
$literal expn
An expression of type text or data covered by a base register defined in a ".using" directive.

The following symbols are used to show the assembled result. A character repeated indicates that the field is wider that one hex digit.

a,b,c Registers ra, rb, and rc.

f A register bit position.

n A numeric field.

d A displacement from a register or the current location.

Most numeric fields and displacements represent sign-extended two's complement quantities. In the Operations column of the following table, "(unsigned)" indicates instructions that do not sign-extend.

Source Syntax		Assembled Format	Operation
a	ra,rb	e1ab	Add
abs	ra,rb	e0ab	Absolute
ae	ra,rb	f1ab	Add Extended
aei	ra,rb,i	d1ab nnnn	Add Extended Immediate
ai	ra, [rb,] i		(Macro) See Section 8.5
ail	ra,rb,i	c1ab nnnn	Add Immediate Long
ais	ra,i	90an	Add Immediate Short
bala	lbl	8ann nnnn	Branch and Link Absolute (unsigned)
balax	lbl	8bnn nnnn	Branch and Link Absolute with Execute (unsigned) **
bali	ra,lbl	8cad dddd	Branch and Link Immediate
balix	ra,lbl	8dad dddd	Branch and Link Immediate with Execute **
balr	ra,rb	ecab	Branch And Link Register
balrx	ra,rb	edab	Brand And Link Register with Execute **
bb	f,lbl	8efd dddd	Branch on Bit
bbr	f,ra	eefa	Branch on Bit
bbrx	f,ra	effa	Branch on Bit with Execute
bbx	f,lbl	8ffd dddd	Branch on Bit with Execute
bnb	f,lbl	88fd dddd	Branch on Not Bit
bnbr	f,ra	e8fa	Branch on Not Bit
bnbrx	f,ra	e9fa	Branch on Not Bit with Execute
bnbx	f,lbl	89fd dddd	Branch on Not Bit with Execute
c	ra,rb	b4ab	Compare
ca16	ra,rb	f3ab	Compute Address 16-bit
cal	ra,reloc	c8ab dddd	Compute Address Lower Half
cal16	ra,reloc	c2ab dddd	Compute Address Lower Half 16-bit (unsigned)
cas	ra,rb,rc	6abc	Compute Address Short
cau	ra,reloc	d8ab dddd	Compute Address Upper Half (unsigned)
ci	ra, i		(Macro) See Section 8.5
cil	ra,i	d40a nnnn	Compare Immediate Long
cis	ra,i	94an	Compare Immediate Short
cl	ra,rb	b3ab	Compare Logical
cli	ra, i		(Macro) See Section 8.5
clil	ra,i	d30a nnnn	Compare Logical Immediate Long
clrbl	ra,i	99an	Clear Bit Lower
clrbu	ra,i	98an	Clear Bit Upper
clrsb	ra,i	95an	Clear SCR Bit
clz	ra,rb	f5ab	Count Leading Zeros
d	ra,rb	b6ab	Divide Step
dec	ra,i	93an	Decrement
exts	ra,rb	b1ab	Extend Sign
get*	ra,$expr		(Macro) See Section 8.5
get*	ra,reloc		(Macro) See Section 8.5
inc	ra,i	91an	Increment
ior	ra,reloc	cbab dddd	Input/Output Read (unsigned)
iow	ra,reloc	dbab dddd	Input/Output Write (unsigned)
jb	f,lbl	08dd to 0fdd	Jump on Bit
jnb	f,lbl	00dd to 07dd	Jump on Not Bit

** If a two-byte instruction follows a Branch and Link with Execute, *as* appends a 'jnop'.

Source Syntax		Assembled Format	Operation
l	ra,reloc	cdab dddd	Load
lc	ra,reloc	ceab dddd	Load Character
lcs	ra,reloc	4dab	Load Character Short
lh	ra,reloc	daab dddd	Load Half
lha	ra,reloc	caab dddd	Load Half Algebraic
lhas	ra,reloc	5dab	Load Half Algebraic Short
lhs	ra,0(rb)	ebab	Load Half Short
lis	ra,i	a4an	Load Immediate Short
load*	ra,expr[(rb)]		(Macro) See Section 8.5
lm	ra,reloc	c9ab dddd	Load Multiple
lps	i,reloc	d0nb dddd	Load Program Status
ls	ra,reloc	7dab	Load Short
m	ra,rb	e6ab	Multiply Step
mc03	ra,rb	f9ab	Move Character 0 from 3
mc13	ra,rb	faab	Move Character 1 from 3
mc23	ra,rb	fbab	Move Character 2 from 3
mc30	ra,rb	fdab	Move Character 3 from 0
mc31	ra,rb	feab	Move Character 3 from 1
mc32	ra,rb	ffab	Move Character 3 from 2
mc33	ra,rb	fcab	Move Character 3 from 3
mfs	ra,rb	96ab	Move From SCR ra to register rb
mftb	ra,rb	bcab	Move From Test Bit
mftbil	ra,i	9dan	Move From Test Bit Immediate Lower
mftbiu	ra,i	9can	Move From Test Bit Immediate Upper
mr	ra,rb		(Macro) See Section 8.5
mts	ra,rb	b5ab	Move To SCR ra from register rb
mttb	ra,rb	bfab	Move To Test Bit
mttbil	ra,i	9fan	Move To Test Bit Immediate Lower
mttbiu	ra,i	9ean	Move To Test Bit Immediate Upper
n	ra,rb	e5ab	And
ni	ra,rb,i		(Macro) See Section 8.5
nilo	ra,rb,i	c6ab nnnn	And Immediate Lower Half Extended Ones (unsigned)
nilz	ra,rb,i	c5ab nnnn	And Immediate Lower Half Extended Zeros (unsigned)
niuo	ra,rb,i	d6ab nnnn	And Immediate Upper Half Extended Ones (unsigned)
niuz	ra,rb,i	d5ab nnnn	And Immediate Upper Half Extended Zeros (unsigned)
o	ra,rb	e3ab	Or
oi	ra,rb,i		(Macro) See Section 8.5
oil	ra,rb,i	c4ab nnnn	Or Immediate Lower Half (unsigned)
oiu	ra,rb,i	c3ab nnnn	Or Immediate Upper Half (unsigned)
onec	ra,rb	f4ab	Ones' Complement
put*	ra,reloc		(Macro) See Section 8.5
s	ra,rb	e2ab	Subtract
sar	ra,rb	b0ab	Shift Algebraic Right
sari	ra,i	a0an	Shift Algebraic Right Immediate
sari16	ra,i	a1an	Shift Algebraic Right Immediate plus 16
se	ra,rb	f2ab	Subtract Extended
setbl	ra,i	9ban	Set Bit Lower

Source Syntax		Assembled Format	Operation
setbu	ra,i	9aan	Set Bit Upper
setsb	ra,i	97an	Set SCR Bit
sf	ra,rb	b2ab	Subtract From
sfi	ra,rb,i	d2ab nnnn	Subtract From Immediate
shl	ra,i		(Macro) See Section 8.5
shla	ra,i		(Macro) See Section 8.5
shr	ra,i		(Macro) See Section 8.5
shra	ra,i		(Macro) See Section 8.5
si	ra,[rb,]i		(Macro) See Section 8.5
sil	ra,rb,i		(Macro) See Section 8.5
sis	ra,i	92an	Subtract Immediate Short
sl	ra,rb	baab	Shift Left
sli	ra,i	aaan	Shift Left Immediate
sli16	ra,i	aban	Shift Left Immediate plus 16
slp	ra,rb	bbab	Shift Left Paired
slpi	ra,i	aean	Shift Left Paired Immediate
slpi16	ra,i	afan	Shift Left Paired Immediate plus 16
sr	ra,rb	b8ab	Shift Right
sri	ra,i	a8an	Shift Right Immediate
sri16	ra,i	a9an	Shift Right Immediate plus 16
srp	ra,rb	b9ab	Shift Right Paired
srpi	ra,i	acan	Shift Right Paired Immediate
srpi16	ra,i	adan	Shift Right Paired Immediate plus 16
st	ra,reloc	ddab dddd	Store
stc	ra,reloc	deab dddd	Store Character
stcs	ra,reloc	1dab	Store Character Short
sth	ra,reloc	dcab dddd	Store Half
sths	ra,reloc	2dab	Store Half Short
stm	ra,reloc	d9ab dddd	Store Multiple
store*	ra,expr[(rb)],rc		(Macro) See Section 8.5
sts	ra,reloc	3dab	Store Short
svc	abs(ra)	c00a nnnn	Supervisor Call (unsigned)
tgte	ra,rb	bdab	Trap if Register Greater Than or Equal
ti	f,ra,i	ccfa nnnn	Trap on Condition Immediate
tlt	ra,rb	beab	Trap if Register Less Than
tsh	ra,reloc	cfab dddd	Test and Set Half
twoc	ra,rb	e4ab	Two's Complement
wait		f000	Wait
x	ra,rb	e7ab	Exclusive Or
xi	ra,rb,i		(Macro) See Section 8.5
xil	ra,rb,i	c7ab nnnn	Exclusive Or Immediate Lower Half (unsigned)
xiu	ra,rb,i	d7ab nnnn	Exclusive Or Immediate Upper Half (unsigned)

8.2. Extended Mnemonics: Branch on Bit

Source Syntax		Assembled Format	Operation
b	lbl	888d dddd	Branch
bc0	lbl	8ecd dddd	Branch on Carry 0
be	lbl	8ead dddd	Branch on Equal
beq	lbl	8ead dddd	Branch on Equal
bh	lbl	8ebd dddd	Branch on High
bhe	lbl	889d dddd	Branch on High or Equal
bl	lbl	8e9d dddd	Branch on Low
ble	lbl	88bd dddd	Branch on Low or Equal
bm	lbl	8e9d dddd	Branch on Minus
bnc0	lbl	88cd dddd	Branch on Not Carry 0
bne	lbl	88ad dddd	Branch on Not Equal
bnh	lbl	88bd dddd	Branch on Not High
bnl	lbl	889d dddd	Branch on Not Low
bnm	lbl	889d dddd	Branch on Not Minus
bno	lbl	88ed dddd	Branch on Not Overflow
bnp	lbl	88bd dddd	Branch on Not Plus
bntb	lbl	88fd dddd	Branch on Not Test Bit
bnz	lbl	88ad dddd	Branch on Not Zero
bo	lbl	8eed dddd	Branch on Overflow
bp	lbl	8ebd dddd	Branch on Plus
btb	lbl	8efd dddd	Branch on Test Bit
bz	lbl	8ead dddd	Branch on Zero
nop	lbl	8eod dddd	No Operation
bc0x	lbl	8fcd dddd	Branch on Carry 0 with Execute
beqx	lbl	8fad dddd	Branch on Equal with Execute
bex	lbl	8fad dddd	Branch on Equal with Execute
bhex	lbl	899d dddd	Branch on High or Equal with Execute
bhx	lbl	8fbd dddd	Branch on High with Execute
blex	lbl	89bd dddd	Branch on Low or Equal with Execute
blx	lbl	8f9d dddd	Branch on Low with Execute
bmx	lbl	8f9d dddd	Branch on Minus with Execute
bnc0x	lbl	89cd dddd	Branch on Not Carry 0 with Execute
bnex	lbl	89ad dddd	Branch on Not Equal with Execute
bnhx	lbl	89bd dddd	Branch on Not High with Execute
bnlx	lbl	899d dddd	Branch on Not Low with Execute
bnmx	lbl	899d dddd	Branch on Not Minus with Execute
bnox	lbl	89ed dddd	Branch on Not Overflow with Execute
bnpx	lbl	89bd dddd	Branch on Not Plus with Execute
bntbx	lbl	89fd dddd	Branch on Not Test Bit with Execute
bnzx	lbl	89ad dddd	Branch on Not Zero with Execute
box	lbl	8fed dddd	Branch on Overflow with Execute
bpx	lbl	8fbd dddd	Branch on Plus with Execute
btbx	lbl	8ffd dddd	Branch on Test Bit with Execute
bx	lbl	898d dddd	Branch with Execute
bzx	lbl	8fad dddd	Branch on Zero with Execute
nopx	lbl	8f8d dddd	No Operation with Execute

8.3. Extended Mnemonics: Branch on Bit Register

Source Syntax		Assembled Format	Operation
bc0r	ra	eeca	Branch on Carry 0
beqr	ra	eeaa	Branch on Equal
ber	ra	eeaa	Branch on Equal
bher	ra	e89a	Branch on High or Equal
bhr	ra	eeba	Branch on High
bler	ra	e8ba	Branch on Low or Equal
blr	ra	ee9a	Branch on Low
bmr	ra	ee9a	Branch on Minus
bnc0r	ra	e8ca	Branch on Not Carry 0
bner	ra	e8aa	Branch on Not Equal
bnhr	ra	e8ba	Branch on Not High
bnlr	ra	e89a	Branch on Not low
bnmr	ra	e89a	Branch on Not Minus
bnor	ra	e8ea	Branch on Not Overflow
bnpr	ra	e8ba	Branch on Not Plus
bntbr	ra	e8fa	Branch on Not Test Bit
bnzr	ra	e8aa	Branch on Not Zero
bor	ra	eeea	Branch on Overflow
bpr	ra	eeba	Branch on Plus
br	ra	e88a	Branch
btbr	ra	eefa	Branch on Test Bit
bzr	ra	eeaa	Branch on Zero
nopr	ra	ee8a	No Operation
bc0rx	ra	efca	Branch on Carry 0 with Execute
beqrx	ra	efaa	Branch on Equal with Execute
berx	ra	efaa	Branch on Equal with Execute
bherx	ra	e99a	Branch on High or Equal with Execute
bhrx	ra	efba	Branch on High with Execute
blerx	ra	e9ba	Branch on Low or Equal with Execute
blrx	ra	ef9a	Branch on Low with Execute
bmrx	ra	ef9a	Branch on Minus with Execute
bnc0rx	ra	e9ca	Branch on Not Carry 0 with Execute
bnerx	ra	e9aa	Branch on Not Equal with Execute
bnhrx	ra	e9ba	Branch on Not High with Execute
bnlrx	ra	e99a	Branch on Not Low with Execute
bnmrx	ra	e99a	Branch on Not Minus with Execute
bnorx	ra	e9ea	Branch on Not Overflow with Execute
bnprx	ra	e9ba	Branch on Not Plus with Execute
bntbrx	ra	e9fa	Branch on Not Test Bit with Execute
bnzrx	ra	e9aa	Branch on Not Zero with Execute
borx	ra	efea	Branch on Overflow with Execute
bprx	ra	efba	Branch on Plus with Execute
brx	ra	e98a	Branch with Execute
btbrx	ra	effa	Branch on Test Bit with Execute
bzrx	ra	efaa	Branch on Zero with Execute
noprx	ra	ef8a	No Operation but with Execute

8.4. Extended Mnemonics: Jump

The operand field consists of a label defined in the same text or data segment as the jump instruction, and located within -256 to +254 bytes.

Source Syntax		Assembled Format	Operation
j	lbl	00dd	Jump
jc0	lbl	0cdd	Jump on Carry 0
je	lbl	0add	Jump on Equal
jeq	lbl	0add	Jump on Equal
jh	lbl	0bdd	Jump on High
jhe	lbl	01dd	Jump on High or Equal
jl	lbl	09dd	Jump on Low
jle	lbl	03dd	Jump on Low or Equal
jm	lbl	09dd	Jump on Minus
jnc0	lbl	04dd	Jump on Not Carry 0
jne	lbl	02dd	Jump on Not Equal
jnh	lbl	03dd	Jump on Not High
jnl	lbl	01dd	Jump on Not Low
jnm	lbl	01dd	Jump on Not Minus
jno	lbl	06dd	Jump on Not Overflow
jnop	lbl	08dd	No Operation
jnp	lbl	03dd	Jump on Not Positive
jntb	lbl	07dd	Jump on Not Test Bit
jnz	lbl	02dd	Jump on Not Zero
jo	lbl	0edd	Jump on Overflow
jp	lbl	0bdd	Jump on Positive
jtb	lbl	0fdd	Jump on Test Bit
jz	lbl	0add	Jump on Zero

8.5. Macro Instructions

The macro instructions generate different instruction sequences depending upon the value of an operand:

sil ra,rb,i

generates an 'ail' with the value of i negated; i must be between −32767 and 32768.

mr ra,rb

generates a 'cas' with r0 as the third operand.

ai ra, [rb,] i
si ra, [rb,] i
ci ra, i
cli ra, i

generates a long or short format instruction depending upon the value of i, and substitutes ra for an omitted rb.

ni ra,rb,i

gives the effect of an and with a 32-bit i by generating a sequence of one or two 'niuz', 'niuo', 'nilz', and 'nilo' instructions.

xi ra,rb,i

gives the effect of an exclusive or with a 32-bit i by generating 'xiu', 'xil', 'xiu' and 'xil', or 'cal' and 'x'.

oi ra,rb,i

gives the effect of an inclusive or with a 32-bit i by generating 'oiu', 'oil', 'oiu' and 'oil', or 'cal' and 'o'.

shl ra,i
shla ra,i

generates a 'sli' or 'sli16', depending on i. i must be in 0-31.

shr ra,i

generates a 'sri' or 'sri16', depending on the value of i. i must be in 0-31.

shra ra,i

generates a 'sari' or 'sari16', depending on the value of i. i must be in 0-31.

get ra, reloc
getha ra, reloc
geth ra, reloc
getc ra, reloc
put ra, reloc
puth ra, reloc
putc ra, reloc

generates a storage reference instruction in long or short form depending on the value of the displacement.

The following macros facilitate generating address constants, and loading and storing in arbitrary memory locations, by exploiting split address relocation. (See *a.out*(5).)

get ra, $expr[(rb)]
getha ra, $expr
geth ra, $expr
getc ra, $expr

If the optional index **(rb)** is present, *as* generates a 'cau' and 'cal'. Otherwise, for an absolute $expr, *as* generates a 'lis', 'cal', 'cal16', or 'cal16' and 'oiu', depending upon the value of expr. For a relocatable or external $expr, *as* generates a 'cal16' and 'oiu'.

load ra, expr[(rb)]
load ra, expr[(rb)]
loadh ra, expr[(rb)]
loadha ra, expr[(rb)]
loadc ra, expr[(rb)]

As generates a 'cau ra' followed by 'l,' 'lh,' 'lha,' or 'lc'. expr may be absolute, relocatable, or external. *ra* may not be r0.

store ra, expr[(rb)],rc
storeh ra, expr[(rb)],rc
storeha ra, expr[(rb)],rc
storec ra, expr[(rb)],rc

As generates a 'cau rc' followed by 'st', 'sth', or 'stc'. expr may be absolute, relocatable, or external. *rc* is a temporary register and may not be r0. storeha is equivalent to storeh.

9. DIAGNOSTICS

Diagnostics are written to standard output. They are intended to be self-explanatory and report errors and warnings. Error diagnostics complain about lexical, syntactic and some semantic errors, and abort the assembly.

The assembler may abandon a statement in error and continue processing sometimes on the same line, sometimes on the next. The result is that one error may lead to spurious diagnostic messages and sometimes "phase errors" where a label has a changed value in the second pass.

10. LIMITS

limit	what
arbitrary[1]	Files to assemble
BUFSIZ	Significant characters per name
arbitrary	Characters per input line
arbitrary	Characters per string
arbitrary	Symbols
4	Text segments
4	Data segments

The number of tokens in a literal definition is limited by the size of the tokenized literal (i.e. by the size of the literal after it has been scanned by the assembler to form a string of tokens). The effective limit is approximately twenty terms in one literal expression.

[1]Although the number of characters available to the *argv* line is restricted by UNIX operating systems to 10240.

This page intentionally left blank.

Floating Point Arithmetic

ABSTRACT

This article describes floating point arithmetic in IBM/4.3. The article includes the following sections:

1. **Comparison with Vax F- and D-Format Arithmetic**
2. **Compatibility with Previous Releases**
3. **Floating Point Hardware**

Floating point arithmetic in IBM/4.3 conforms to IEEE Standard 754 for binary floating point arithmetic. Single-and double representations are supported.

1. Comparison with F- and D-Format Arithmetic

IEEE arithmetic produces results that in general are at least as accurate as those from IBM System/370 arithmetic. Single precision is very similar to VAX F-format in range and precision. Double precision is comparable to VAX D-format; see (1) below.

The salient differences from the F- and D-format arithmetic used in C and 4.3BSD on the VAX are as follows:

(1) Type double has a mantissa of 53 bits rather than 56; the exponent range is approximately 3e-308 to 1e308, rather than 3e-39 to 1e38. Magnitudes as small as 3e-324 are represented with reduced precision.

(2) IEEE arithmetic includes representations for plus and minus infinity and a collection of "Not-a-Number" (NaN) values. *Printf*(3S) represents these on output as INF and NAN(). Signed zero values are also supported; $+0 = -0$, but $1/-0 = -\text{INF}$.

(3) Rounding modes and exception handling are supported; user code can change the settings via library functions; see *ieee*(3) and *ecvt*(3). IEEE default settings are in force initially: the rounding mode is round to nearest; on an exception, proceed without trap (that is, return a reasonable result).

(4) With the default exception handling, several arithmetic operations that signal SIGFPE on the VAX do not on the IBM RT PC. Exponent overflow receives IEEE default handling, which is to return infinity. Other values larger than 1e38 are represented correctly rather than overflowing. 0/0, INF/INF and certain other operations produce NaNs, which will propagate through subsequent arithmetic operations. Library functions that signaled SIGFPE, however, continue to do so.

(5) VAX F and D formats differ only in mantissa width: the first word in D-format has the same interpretation as an F-format number. Consequently, on a VAX, type mismatches can produce plausible incorrect results, differing from the correct results by one part in a million. IEEE single and double formats differ in exponent width as well as mantissa width, so type mismatches (from nonportable unioning, function calls, or using "%e" for "%le" in *scanf*(3S), for instance) generally produce answers that are dramatically, rather than subtly, wrong.

(6) The IEEE recommended functions are supported; see *ieee*(3) for details.

Also, two new functions are provided to perform the IEEE required operations of round floating-point number to integral value (according to the current rounding mode) and floating-point remainder. These are *rint* and *drem* (see *ieee*(3)).

2. Compatibility with Previous Releases

Note that while this initial release of the new IBM/4.3 Floating Point support gives the maximum compatibility possible, future releases may not. Most a.outs compiled and linked under previous releases will produce the same results when run under this release. However, performance will be improved by recompiling, especially if running on an RT with an APC.

2.1. A.outs Linked with -lfpa Option

The **-lfpa** option, in previous releases, was intended for use when the FPA was the only supported floating point hardware. This new support eliminates the need for the flag. Executables (a.outs) previously linked with **-lfpa** will not run on a machine *with* an APC card and *without* an FPA. These executables should be recompiled and relinked.

For those systems where users have many Makefiles, scripts, and so forth, that depend on the **-lfpa** flag, the system administrator can install a dummy library to satisfy the loader. A dummy library is provided for this purpose in */usr/src/old/fpa*.

2.2. Linking Old and New Object Files (.o's)

For best performance, object files linked to one another should be recompiled under IBM/4.3 so that all modules are using the same support. If you choose not to recompile, *ld*(1) will print a warning message. The resulting executable (a.out) will use the FPA (if it is present) or the emulator (if the FPA is not present).

3. Floating Point Hardware

Floating point operations can be performed by the following types of hardware:

FPA The first Floating Point Accelerator for the RT (sometimes called the FPA I) supports both single and double precision.

AFPA The second, or Advanced, Floating Point Accelerator for the RT (sometimes called the FPA II) supports both single and double precision.

MC881 The Motorola 68881 on the Advanced Processor Card (APC) supports extended as well as single and double precision (but the latter two cause a performance degradation). Extended precision is the default. The MC881 offers the fastest performance.

In the absence of floating point hardware, RT floating point instructions can be executed via an emulation package (which performs the same computations in software as the FPA).

Floating point support is chosen in the following order, if available:

(1) MC881
(2) AFPA
(3) FPA
(4) Emulator

To force the use of one of the above, set the environment variable **FPA** to *mc881*, *afpa*, *fpa*, or *emul*. If the named hardware is available, that support will be chosen rather than the default.

Due to the 68881's higher default precision, there may be a slight difference in results for floating point instructions executed via the 68881 and the FPAs or emulator. For example, intermediate results left in extended precision during calculation:

```
a = b*c-d
```

may cause "a" to differ slightly from "a" computed as:

```
t = b*c
a = t-d
```

Furthermore, register variables are left in extended precision in the MC881 and in single or double precision in the FPA, AFPA, and emulator.

To offer the best performance for floating point instructions, IBM/4.3 by default uses the fastest hardware available, and the "fastest" precision for that hardware (depending upon operation and type of arguments). To accommodate the need for predictable results regardless

of hardware or software used, IBM/4.3 provides a new environment variable, **FP_PRECISION**, with four options:

fast (default)

precise (use widest possible precision)

double (round all operations to double)

single (round all single operations to single and all others to double; used rarely but required by the IEEE 754 Standard)

Double and single modes provide cross-hardware conformance. That is, the results of a floating point instruction performed in single (or double) mode are identical, whether the instruction is performed using the MC881, FPAs, or emulator.

Note, however, that forcing the precision may seriously degrade performance. The following table summarizes the effect of the precision mode on the generated code:

mode	68881	FPAs (AFPA, FPA, Emulator)
fast	extended ops on all (this implies extended math)	single ops on single args double ops on double args (this implies double math)
precise	extended ops on all (this implies extended math)	double ops on all (this implies double math)
double	68881 mode set to double (this implies double math)	double ops on all (this implies double math)
single	68881 mode set to double double ops on double args (this implies double math) single ops on single args (set mode to single for op, then set it back to double)	single ops on single args double ops on double args (this implies double math)

See "IBM/4.3 Linkage Convention" in Volume II, Supplementary Documents for more information.

The C Subroutine Interface for the IBM Academic Information Systems Experimental Display

ABSTRACT

This paper describes a subroutine interface for the IBM Academic Information Systems experimental display transported for use under the C programming language and IBM/4.3. It contains the following chapters and appendices:

1. **Introduction** contains some background information on the experimental display.
2. **Controlling the Interface** describes the subroutines that control the interface session.
3. **Setting Graphics Parameters** describes the subroutines that set graphics parameters. Graphics parameters modify the way in which subroutines that update the screen operate.
4. **Querying Graphics Parameters** describes the subroutines that return the current values of graphics parameters.
5. **Issuing Graphics Primitives** describes the subroutines that build orders that update the screen.
6. **Controlling the Cursor** describes the subroutines that enable programs to control the experimental display cursor.
7. **Defining Fonts** describes the orders that control the experimental display font mechanism.
8. **Manipulating Fonts** describes the subroutines that manipulate fonts.

Appendix A describes the format of a font file.

Appendix B describes character definitions.

Appendix C describes *aedjournal*(1) and *aedrunner*(1), supplied programs which display and run commands in a log file.

Appendix D describes the examples supplied with the subroutine interface.

1. INTRODUCTION

The experimental display is a black-and-white, all-points-addressable, bit-mapped display that attaches to the IBM RT PC. The experimental display features 819,200 points on the screen, each one individually selectable. The experimental display adapter contains a very fast on-board processor that allows text and graphics to be drawn at a rate much faster than the host alone would allow. The experimental display processor is programmed to accept high-level orders from the host, and to present the results on the screen.

The characteristics of communicating with the experimental display are determined by the microprogram running in the experimental display adapter processor. This program is stored in writable control store and is loadable from the host.

The interface described in this paper is a set of functions designed to support a window manager, and is composed primarily of subroutines, as distinguished from functions. A typical subroutine uses parameters to receive input as well as to return output. C passes parameters by value; to call a subroutine which returns information, you must supply an address for the returning value as the parameter.

Calls that supply an *address* for return in this package should usually supply the address of a *short* (16-bit) integer. Calls that pass integer *values* can usually get by with either *short* or *int*. See the individual routines.

Many of the subroutines do return a value as a function would. Generally, values are used for error return codes and special case handling. It is strongly recommended that applications monitor return codes in order to prevent bizarre events and possibly more severe errors.

When linking, you must specify *-laed* to pick up the experimental display library.

2. CONTROLLING THE INTERFACE

This chapter describes the subroutines that control the interface.

2.1. VI_Init: Initialize the Subroutine Interface

VI_Init initializes the experimental display and returns the dimensions of the screen. Current display models are 1024 bits wide by 800 bits high. The top left point is (0,0) and the bottom right point is (1023,799). A 16-bit word used as an image on the experimental display will have its least significant bits to the right. */usr/lib/aed/whim.aed* must be accessible at run time.

Because *VI_Init* initializes the experimental display, it should be called before the other routines of the package.

VI_Init has the following format:

```
VI_Init(wd,ht)
  short *wd,*ht;          /* screen dimensions */
```

2.2. VI_Force: Force Output of Graphics Orders

Commands built with subroutines described in "Setting Graphics Parameters" and "Issuing Graphics Primitives" later in this paper generally do not send their output to the screen immediately. Instead the output remains in a buffer until the buffer is full, when its output is sent to the screen. Use *VI_Force* to force output in the current buffer to be transmitted before the buffer is full.

VI_Force has the following format:

```
VI_Force()
```

2.3. VI_Login: Begin Logging Subroutine Calls

VI_Login specifies that subsequent subroutine calls are to be echoed into the specified file. If a log file is already open, *VI_Login* closes it before opening the new file; *VI_Login* overwrites an existing file. All orders to the experimental display are logged until a logout call (*Logout*) is issued. The log file may later be executed from within a program using *VI_Run* or on its own using *aedrunner*(1). It may also be examined with *aedjournal*(1). (Appendix C of this paper describes these programs.) *VI_Login* returns a negative value if there is an error, and a nonnegative value if the call is successful.

VI_Login has the following format:

```
int VI_Login(filename)
    char *filename;  /* file to log to */
```

2.4. VI_Logout: Close a Log File

VI_Logout closes the log file and returns one of three values:

Value	Meaning
0	Normal completion
-1	Error in closing file
-2	No file found to close

VI_Logout has the following format:

```
int VI_Logout()
```

2.5. VI_Run: Process a Log File

VI_Run executes the commands logged in the specified file; *filename* is the name of a log file that was created by *VI_Login*. Using *VI_Run* with a log file has the same effect of executing *aedrunner*(1) from within a program, allowing a series of orders which require much calculation to be figured only once, logged, then quickly retrieved when needed. *VI_Run* returns 0 for a normal completion, and -1 for an error condition.

VI_Run has the following format:

```
int VI_Run(filename)
    char *filename;          /* log file name */
```

2.6. VI_Term: Terminate the Subroutine Interface

VI_Term completes processing, closes the log file, and forces transmission of the graphics buffer to the experimental display.

VI_Term has the following format:

```
VI_Term()
```

3. SETTING GRAPHICS PARAMETERS

Graphics parameters modify the way in which the primitives described later in this paper operate. This chapter describes the subroutines that set graphic parameters. The initial values of these parameters are:

Clipping window	The clipping window is set to the whole screen.
Screen color	The screen color is white 1's on black 0's, color 0.
Dash pattern	The line dash pattern is solid 1's.
Font	The font is 0. No font is selected.
Merge mode	The merge mode is 12, for replace mode. Data bits replace screen bits.
Line width	Line width is 1.

3.1. VI_Clip: Set Clipping Window

VI_Clip specifies that subsequent primitives drawn on the screen are to be clipped to the specified area. It is the user's responsibility to ensure the sensibility of the window definition.

VI_Clip has the following format:

```
VI_Clip(lx,ly,hx,hy)
  int lx,ly;    /* top left corner of clipping area */
  int hx,hy;    /* bottom right corner of area */
```

3.2. VI_Color: Change Screen Color

VI_Color sets the color of the screen to the specified value: 0 means that bits having the binary value "0" will be black on the screen; 1 means that bits having the binary value "1" will be black on the screen. If this value is different from the previous value, the screen will be inverted, so as to make the change transparent to the application.

VI_Color has the following format:

```
VI_Color(color)
  int color;      /* new color, true for white */
```

3.3. VI_Dash: Set Line Dash Pattern

If no dash pattern has been set, lines drawn with the *VI_RLine* and *VI_ALine* subroutines described in "Issuing Graphics Primitives" are solid lines of 1's. If a pattern has been set, the bits of the pattern word are used in sequence whenever the vector generator would normally output a 1. Setting a pattern of 0x5555 produces a very acceptable dotted line. Other patterns may be used to vary the size of dashes in the line. The length of the pattern can range from 1 to 16 bits. The pattern bits should be left-justified. Setting the pattern length to 0 specifies a return to solid lines.

VI_Dash has the following format:

```
VI_Dash(dash,dashlen)
  unsigned short dash;        /* dash pattern */
  short dashlen;        /* dash pattern length */
```

3.4. VI_Font: Select Font

The current font affects the results of the *VI_String* primitive described under "Issuing Graphics Primitives." Font IDs range from 0 to 255 and are returned by calls to *VI_GetFont*. See "Defining Fonts" later in this paper for more information.

VI_Font has the following format:

```
VI_Font(fontid)
  int fontid;              /* font ID */
```

3.5. VI_Merge: Set Merge Mode

The merge mode is a number from 0 to 15 that specifies how the bits generated by primitives are to be combined with bits already on the screen. The merge mode is simply an encoding of the logical function used to combine screen bits and data bits. Encoding the desired result of each of the combinations in the table below generates the merge mode that should be used to get that effect. For example, to *or* the data you are adding with the data already on the screen, you would use a merge mode of 14:

Data Bit	1	1	0	0	
Screen Bit	1	0	1	0	
Example: OR mode	1	1	1	0	= 14

VI_Merge has the following format:

```
VI_Merge(merge)
  int merge;               /* merge mode */
```

3.6. VI_Width: Set Line Width

VI_Width specifies a value between 1 and 16 that is to be the line width. Normally, lines are 1 bit thick.

VI_Width has the following format:

```
VI_Width(width)
  int width;               /* line width */
```

4. QUERYING GRAPHICS PARAMETERS

The subroutines in this chapter return the current values of the graphics parameters described above. Each subroutine requires an address in which to store the value to be returned. All of these subroutines force transmission of graphics data in the current buffer.

4.1. VI_QClip: Query Clipping Rectangle

VI_QClip returns the current clipping rectangle.

VI_QClip has the following format:

```
VI_QClip(lx,ly,hx,hy)
 short *lx,*ly;      /*top left corner of clipping area*/
 short *hx,*hy;          /* bottom right corner */
```

4.2. VI_QColor: Query Current Color

VI_QColor returns the current color of the screen: 0 means that bits having the binary value "0" will be black on the screen; 1 means that bits having the binary value "1" will be black on the screen.

VI_QColor has the following format:

```
VI_QColor(color)
 short *color; /* current color, true for white */
```

4.3. VI_QDash: Query Dash Pattern

VI_QDash returns the current line dash pattern in the format described for *VI_Dash*. If *dashlen* is 0, the lines are solid.

VI_QDash has the following format:

```
VI_QDash(dash,dashlen)
 unsigned short *dash;          /* dash pattern */
 short *dashlen;      /* length of dash pattern */
```

4.4. VI_QFont: Query Font

VI_QFont returns the ID and name of the current font. The font ID is 0 if no font has been set. The pointer *fontname* should point to a block of characters large enough to hold a file name (including an extension) on your operating system, along with a string-termination byte. If you know beforehand the size of your file name, you may allow only as many bytes as required. Be aware of the string-terminator byte; there must be room for it.

VI_QFont has the following format:

```
VI_QFont(fontid,fontname)
 short *fontid;            /* current font ID */
 char *fontname;      /* current font name */
```

4.5. VI_QMerge: Query Merge Mode

VI_QMerge returns the current merge mode in the format described for the *VI_Merge* subroutine described in "Setting Graphics Parameters."

VI_QMerge has the following format:

```
VI_QMerge(merge)
 short *merge;          /* current merge mode */
```

4.6. VI_QPoint: Query Current Point

VI_QPoint returns the location of the current point. This command is especially useful after a *VI_String* primitive has been issued, since character definitions can change the current point in unpredictable ways.

VI_QPoint has the following format:

```
VI_QPoint(x,y)
  short *x,*y;              /* current point */
```

4.7. VI_QWidth: Query Line Width

VI_QWidth returns the current line width as a number between 1 and 16.

VI_QWidth has the following format:

```
VI_QWidth(width)
  short *width;             /* line width */
```

5. ISSUING GRAPHICS PRIMITIVES

This chapter describes the subroutines that build orders that update the screen. Orders are transmitted only when the buffer is full, when specified with *VI_Force*, or when other non-graphics subroutines are called.

The graphics primitives work in screen coordinates: *x* represents the horizontal axis on the screen, and increases to the right; *y* represents the vertical axis and increases to the bottom of the screen. The coordinates (0,0) represent the top-left corner of the screen. Subroutines will accept coordinates that are off the screen; the behavior is as if there were a clipping window the size of the screen in a larger universe.

Several of the primitives depend on the current point. This point is initially set to (0,0) and can be modified by primitives.

5.1. VI_AMove: Move the Current Point to an Absolute Location

VI_AMove moves the current point to the specified coordinates. No change is made to the screen.

VI_AMove has the following format:

```
VI_AMove(x,y)
  int x,y;                /* new point */
```

5.2. VI_RMove: Move the Current Point to a Relative Location

VI_RMove moves the current point by the specified displacement. No change is made to the screen.

VI_RMove has the following format:

```
VI_RMove(dx,dy)
  int dx,dy;    /* displacement from old point */
```

5.3. VI_ALine: Draw a Line with an Absolute Location

VI_ALine draws a line from the current point to the specified point (the line's end point) according to the current values of the width and dash pattern parameters. A line is normally of 1's, and is merged with the window data according to the current merge mode. The specified point becomes the current point.

VI_ALine has the following format:

```
VI_ALine(x,y)
  int x,y;                /* end point of line */
```

5.4. VI_RLine: Draw a Line with a Relative Location

VI_RLine draws a line from the current point to the current point displaced by the specified values, according to the current values of the width and dash pattern parameters. A line is normally of 1's, and is merged with the window data according to the current merge mode. The current point is incremented by the displacement.

VI_RLine has the following format:

```
VI_RLine(dx,dy)
  int dx,dy;      /* displacement to endpoint */
```

5.5. VI_Circle: Draw a Circle

VI_Circle draws a circle with the specified radius and the current point as its center. The current point is unchanged.

VI_Circle has the following format:

```
VI_Circle(radius)
 int radius;            /* circle radius */
```

5.6. VI_MImage: Draw an Image from Memory

VI_MImage draws an image of the specified dimensions whose top left corner is at the current point. The current point is not changed.

Data must be the first byte of an image large enough to fill the rectangle specified by *wd* and *ht*, or an addressing error may result. The image data should be in scanline order, from top to bottom, with each scanline padded to the next 16-bit word. For example, for a width of WD and height of HT, there should be 2*HT*(WD+15)/16 bytes of image data.

VI_MImage has the following format:

```
VI_MImage(wd,ht,data)
 int wd,ht;          /* dimensions of image */
 unsigned short *data;  /* first byte of image */
```

5.7. VI_FImage: Draw an Image from a File

VI_FImage draws the image contained in the specified file, placing its top left corner at the current point. The current point is unchanged.

The image file must have the format shown below. The data words should be in the same format as for the *VI_MImage* subroutine.

Offset (bytes)	Description
0	The width of the image
2	The height of the image
4	Image data

VI_FImage has the following format:

```
VI_FImage(filename)
 char *filename;    /* file name of image to draw */
```

5.8. VI_Tile: Tile a Rectangle

VI_Tile fills a rectangle of the specified dimensions with the specified pattern. The rectangle's top left corner will be at the current point. The tile pattern must follow the rules for images (see the *VI_MImage* subroutine above), and can be of any size. The tile pattern is aligned to multiples of *twd* and *tht*, not to the bounds of the tiled rectangle, so that rectangular subareas of larger figures can be tiled without regard to their bounds, and the tile patterns will match. The current point is unchanged.

A full rectangle black or white fill can be most quickly drawn by requesting a one-by-one tile. Clearly, only all ON or all OFF may be drawn with this method, but any merge mode may be used.

VI_Tile has the following format:

```
VI_Tile(wd,ht,twd,tht,tile)
 int wd,ht;        /* dimensions of rectangle */
 int twd,tht;          /* dimensions of tile */
 unsigned short *tile;  /* first byte of pattern */
```

5.9. VI_String: Draw a String

VI_String draws the specified string at the current point. Since a character definition is really a sequence of other graphics commands (usually *VI_MImage* and *VI_RMove*), the way in which characters are positioned, stepped, and drawn depends on the font definition. Character definitions typically modify the current point. See "Defining Fonts" later in this paper for more information.

VI_String has the following format:

```
VI_String(s)
  char *s;                    /* string to draw */
```

5.10. VI_Copy: Copy an Area

VI_Copy duplicates the rectangle at *sx,sy* with the dimensions *wd,ht* to the point *tx,ty*. The copied bits are merged with the target area using the specified merge mode, not the merge mode set by *VI_Merge*.

Both the source and destination rectangles must be completely on the screen. The current setting of the clipping window is ignored.

VI_Copy has the following format:

```
VI_Copy(sx,sy,tx,ty,wd,ht,merge)
  int sx,sy;              /* source top-left */
  int tx,ty;              /* target top-left */
  int wd,ht;          /* rectangle dimensions */
  int merge;                  /* merge mode */
```

5.11. VI_MRead: Read Display Data into Memory

VI_MRead reads the specified area of the screen into the array passed as *data*. Image bytes are in the same format as expected by *VI_MImage*. If the screen color is white, the bits are inverted on readback to make the data read back independent of screen color. The area to be read must be completely on the screen. The current setting of the clipping window is ignored.

VI_MRead has the following format:

```
VI_MRead(x,y,wd,ht,data)
  int x,y;          /* top-left corner of area */
  int wd,ht;            /* dimensions of area */
  unsigned short *data;  /* first byte of data */
```

5.12. VI_FRead: Read Display Data into a File

VI_FRead reads the specified area of the screen and places it in the specified file. The file has the same format as expected by *VI_FImage*. If the window color is white, data bits are inverted to make the data independent of the screen color. The area to be read must be completely on the screen. The current setting of the clipping window is ignored.

VI_FRead has the following format:

```
VI_FRead(x,y,wd,ht,filename)
  int x,y;          /* top-left corner of area */
  int wd,ht;            /* dimensions of area */
  char *filename;       /* name of file to place image in */
```

6. CONTROLLING THE CURSOR

The following routines allow programs to control the experimental display cursor by defining it, enabling and disabling it, and changing its position. Note that because the experimental display maintains the cursor separately from the display buffer, the cursor does not have to be removed when a graphics primitive intersects its position.

Initially the cursor is transparent and disabled, and is positioned at the center of the screen.

6.1. VI_MDefnCur: Set Cursor Pattern from Memory

VI_MDefnCur sets the cursor as specified. *xoff,yoff* is the displacement of the cursor pattern from the current position of the cursor. For example, a value of (32,32) would center the cursor pattern around the current point.

The cursor pattern itself is a 64-by-64 bit image, with two planes. A 1 in the black plane indicates that that bit of the cursor should be black. A 1 in the white plane indicates that the cursor should be white in that position. If a bit has a 0 in both planes, the cursor is transparent in that position. If a bit is 1 in both planes, the cursor is white.

The two planes are images in the same format as accepted by *VI_MImage*, and must be 64-by-64, or 512 bytes each.

VI_MDefnCur has the following format:

```
VI_MDefnCur(xoff,yoff,black,white)
  int xoff;       /* x offset of cursor center */
  int yoff;       /* y offset of cursor center */
  unsigned short *black;    /*first byte black mask */
  unsigned short *white;    /*first byte white mask */
```

6.2. VI_FDefnCur: Set Cursor Pattern from File

VI_FDefnCur sets the cursor to the definition in the specified file. The file has the following format:

Offset (bytes)	Description
0	XOFF
2	YOFF
4	BLACK bit pattern
516	WHITE bit pattern

See the description of *VI_MDefnCur* for a description of the fields.

VI_FDefnCur has the following format:

```
VI_FDefnCur(filename)
  char *filename;     /* name of cursor definition file */
```

6.3. VI_EnCur: Enable Cursor

VI_EnCur enables the cursor and displays it if it is not already present. Disabling and reenabling the cursor do not affect its position.

VI_EnCur has the following format:

```
VI_EnCur()
```

6.4. VI_DisCur: Disable Cursor

VI_DisCur disables the cursor and removes it from the screen if it is present. Disabling and reenabling the cursor do not affect its pattern or position.

VI_DisCur has the following format:

```
VI_DisCur()
```

6.5. VI_PosnCur: Set Cursor Position

VI_PosnCur moves the cursor to the specified position. The cursor cannot be moved off the screen.

VI_PosnCur has the following format:

```
VI_PosnCur(x,y)
 int x,y;              /* new cursor position */
```

7. DEFINING FONTS

The font mechanism supported by the experimental display is very general. Characters are not simply raster patterns; instead, each character definition is a simple graphics subroutine, able to move the current point, draw images, change the merge mode, etc. The orders that can occur in a character definition are a subset of the orders built by the graphics primitives subroutines. In addition, two orders, *push* and *pop,* control parameters within a character definition.

7.1. Standard Raster Characters

The most typical use of the font mechanism is for standard raster characters. The sequence of orders is similar to the following:

(1) *VI_Image* at the current point.

(2) *VI_RMove* right by the width of the characters.

This example draws all characters down from the current *y* value.

7.2. Raster Character with Baseline Defined for the Font

The next most common use is a raster character with a baseline defined for the font. The sequence of orders would be similar to the following:

(1) *VI_RMove* up by the ascender height (height above baseline).

(2) *VI_Image* at the current point.

(3) *VI_RMove* down and right by the ascender height and character width.

7.3. Stroked Fonts

Stroked fonts can be defined using *VI_RMove* and *VI_RLine* commands. Stroked characters can be mixed freely with raster characters.

7.4. Three-Color Characters

Three-color characters can be defined with a sequence such as the following:

(1) *VI_RMove* to top of character image.

(2) *VI_Merge 2*, which turns off the screen data having the binary value "1", and leaves it unchanged for screen data having the binary value "0".

(3) *VI_Image*, with a pattern that turns off the black bits of the character.

(4) *VI_Merge 14*, OR mode.

(5) *VI_Image*, with a pattern that turns on the white bits.

(6) *VI_RMove* to start of next character.

With this font selected, characters drawn by the *VI_String* command would draw black, white and transparent patterns, suitable for text drawn over a complex graphics image.

8. MANIPULATING FONTS

Fonts are stored in files, which are loaded into the IBM RT PC memory when requested by applications using the *VI_GetFont* subroutine. Once a font is loaded, it is kept in memory until the program ends, unless explicitly dropped with the *VI_DropFont* subroutine.

8.1. VI_GetFont: Load a Font into Memory

VI_GetFont loads the specified font into memory, if it is not already present. If the font is successfully loaded, the font ID is returned. Setting the current font to this ID with the *VI_Font* routine causes subsequent strings to be displayed in the font. If a font ID of 0 is returned, either the font could not be found, or it did not fit in memory. If the font did not fit in memory, a message will be sent to *stderr*.

VI_GetFont has the following format:

```
VI_GetFont(name,fontid)
  char *name;              /* font name */
  short *fontid;           /* font ID */
```

8.2. VI_DropFont: Release Font

VI_DropFont drops the specified font from memory. The application should not attempt to use the font ID again. If the font is required, a new font ID should be generated by a request to *VI_GetFont*.

VI_DropFont has the following format:

```
VI_DropFont(fontid)
  int fontid;          /* ID of font to release */
```

APPENDIX A. FORMAT OF A FONT FILE

A font definition file begins with an index by character codepoint. The first entry is for codepoint 0x00, the second for 0x01, and so on, up to 0xFF. An index entry has the following format:

Offset	Length in bytes	Description
0	4	Offset of the character definition in the file; an undefined character has an offset of zero.
4	2	Width of inner box of the character.
6	2	Height of inner box of the character.
8	2	Total x displacement caused by character.
10	2	Total y displacement caused by character.
12	2	Distance from the initial x position to the left edge of the inner box.
14	2	Distance from the initial y position to the top edge of the inner box.

A font file consists largely of character definitions, which follow the index. Character definitions do not necessarily appear in order. Undefined characters are not included. Each character definition has the following format:

Offset	Length in bytes	Description
0	2	Character codepoint, in the low byte of the word.
2	2	Length of character definition, in 16-bit words, not including the count. The length of a character definition must be less than 2000 words.
4	count*2	Character definition. A definition consists of a series of orders, as described in Appendix B of this article.

APPENDIX B. CHARACTER DEFINITIONS

Before reading this, you should understand the format of the font file, which contains character definitions, described in Appendix A of this article.

Character definitions consist of a string of orders from the following list. Note that parameter changes made by character definitions do not persist after the character has been completed.

Set Merge Mode

Offset in 16-bit words	Value
0	Merge Command (= 1)
1	Merge Mode

The merge mode is changed to the specified value. The format is the same as described for the *VI_Merge* subroutine.

Set Line Dash Pattern

Offset in 16-bit words	Value
0	Set Dash Command (= 3)
1	Dash Pattern
2	Pattern length

Lines drawn after this command use the specified pattern. A pattern length of zero specifies a return to normal solid lines. The pattern is from 1 to 16 bits, left-justified in the pattern word.

Set Line Width

Offset in 16-bit words	Value
0	Set Width Command (= 4)
1	Line Width

Subsequent lines are drawn with the specified width.

Push Modes

Offset in 16-bit words	Value
0	Push Command (= 12)

The modifiable parameters (merge mode, dash pattern, line width) are pushed onto an internal stack. They may be changed and then later restored with the *pop* order. When a character definition ends, the original modes are restored, regardless of *push* or *pop* orders within a definition.

Pop Modes

Offset in 16-bit words	Value
0	Pop Command (= 13)

The modifiable parameters (merge mode, dash pattern, line width) are restored from the internal stack. When a character definition ends, the original modes are restored, regardless of *push* or *pop* orders within a definition.

Move Relative

Offset in 16-bit words	Value
0	Move Relative Command (= 6)
1	X displacement
2	Y displacement

The indicated displacement is added to the current point. If either coordinate of the current point goes outside the range -32768 to 32767, the value wraps (overflows or underflows).

Draw Line Relative

Offset in 16-bit words	Value
0	Draw Line Relative Command (= 8)
1	X displacement
2	Y displacement

A line is drawn from the current point to the current point plus the displacement. The ending point becomes the new current point. If either coordinate of the current point goes outside the range -32768 to 32767, the value wraps (overflows or underflows).

Draw Circle

Offset in 16-bit words	Value
0	Draw Circle Command (= 14)
1	Circle Radius

A circle with the specified radius is drawn around the current point. The current point is unchanged.

Draw Image

Offset in 16-bit words	Value
0	Draw Image Command (= 9)
1	Image width
2	Image height
3	Image data

The image given is drawn with its top left corner at the current point. The current point is unchanged.

The scanlines of the image must be padded to the next 16-bit word. Thus, the number of words in the image is height*(width + 15) / 16.

Tile Rectangle

Offset in 16-bit words	Value

0	Tile Command (= 10)
1	Rectangle width
2	Rectangle height
3	Tile width
4	Tile height
5	Tile data

The tile image is repeated over the whole area of the indicated rectangle. The tile image data has the same format as data in the *VI_Image* order described above.

APPENDIX C. AEDJOURNAL AND AEDRUNNER

Aedjournal(1) and *aedrunner*(1) are supplied programs which use the interface. Both operate on a log file created with *VI_Login* and *VI_Logout*. *Aedjournal* displays the commands built into the file; *aedrunner* executes those commands.

Debugging with aedjournal

aedjournal file

Although there is no debugging facility as such supplied with this package, you can use *VI_Login* and *VI_Logout* with *aedjournal* to help follow your application program's actions. *Aedjournal* deciphers a file produced by *VI_Login* and reports to standard output all orders passed to the experimental display. Standard output may be redirected as usual. You may inspect this output to discover unintended results.

Beware of the length of logged files. It is very easy to generate thousands of display orders for a seemingly simple picture; thus, try to log the smallest group of orders you believe contains the error. The log routines may be called several times in one application to produce several files of orders, requiring only that each call to *VI_Login* provide a distinct file name.

Executing a log file with aedrunner

aedrunner file ...

Aedrunner executes the orders logged into the specified file, which must have been created with *VI_Login* and *VI_Logout*. *Aedrunner* terminates upon discovery of any error or inconsistency in the file. All additional files which were needed when the log file was constructed must be available in the current directory. Such files are any font, image, or cursor definition files you may have used, and */usr/lib/aed/whim.aed* must exist. Images, cursors, or tiles defined from memory are handled by the log routines and do not require regeneration.

APPENDIX D. SUPPLIED EXAMPLES

All files associated with this package reside in the directory */usr/src/usr.lib/libaed/examples*.

Among the files supplied with the microcode and subroutine library are some source and executable files for you to investigate. The following list includes some of those files, and brief descriptions. It should be easy to figure out the nature of any other files from their names, behavior, or above documentation.

The following programs are copyrighted property of International Business Machines Corporation.

*.fnt	Files with the extension *.fnt* are font files.
showfont	A program that shows a font on the experimental display. The syntax is *showfont filename.*
showfont.c	Source for *showfont.*
zip	A demo that takes up to three parameters. Parameter 1 is number of vectors to remain on the screen. Parameter 2 is minimum delta for each new vector endpoint. It is roughly equivalent to the speed of the zipper. Parameter 3 is maximum delta. The default is *zip 30 2 14.*
zip2	Like *zip* but with two zippers. It takes up to 6 parameters. The default is *zip2 30 2 14 90 1 4.*
zipn	Like *zip* but with 1 to 16 zippers. Parameter 1 is number of zippers. Parameters 2, 3, and 4 are number vectors for zipper 1, minimum delta, and maximum delta. Parameters 5, 6, and 7 are for zipper 2, etc. The default for unspecified zippers is *30, 2,14.* The default is *zipn 1.*
zip.c	Zip source code.
aedrunner.c	*Aedrunner* source code.

This page intentionally left blank.

Programmer's Notes

ABSTRACT

This article is a compendium of insights, suggestions, and notes gathered from the programmers who ported applications to IBM/4.3. The information may save time and frustration for others with the same task.

1. SAMPLE FILES PROVIDED

Four sample files (.login, .cshrc, .logout, and .profile) are provided in /usr/skel. Using these files will simplify initial installation and operation of IBM/4.3.

2. CHARACTER TYPE IS UNSIGNED

Variables of type **char** are unsigned (range 0..255) by default on the RT PC, in contrast to the VAX, where they are signed (range -128..127) by default. With the High C compiler (*hc*(1)), the type **signed char** is available, as well as a command-line option *-Hoff=char_default_unsigned* to make characters signed by default. This option generally produces less efficient code, but can be of value in determining whether signedness is the cause of a bug.

The **unsigned** default uncovers a machine dependency in a common technique for end-of-file testing. In the following program fragment

```
char c;
. . .
if ((c = getchar()) = = EOF) ...
```

the test always fails, since EOF is -1 and c is in 0..255. Declaring c as an **int** is a good machine-independent solution.

With *pcc*(1), there is no type **signed char**, but the following macro might be useful if you need to use an unsigned character as though it were signed:

```
#if '\377' < 0
#define Signed(x) (x)
#else
#define Signed(x) (((x)^128)-128)
#endif
```

3. BYTE ORDERING IS DIFFERENT

The IBM RT PC has sixteen 32-bit general registers. Memory on the IBM RT PC is byte-addressed, but differently than on the VAX.

On the VAX, high order bits are at higher addresses, thus:

```
| - - - w o r d 2 - - - | - - - w o r d 1 - - - | - - - w o r d 0 - - - |
| C 3 , C 2 , C 1 , C 0 | C 3 , C 2 , C 1 , C 0 | C 3 , C 2 , C 1 , C 0 |
| B 3 1 . . . . . B 0 | B 3 1 . . . . . B 0 | B 3 1 . . . . . B 0 |
```

On the IBM RT PC, high order bits are at lower addresses, thus:

```
| - - - w o r d 0 - - - | - - - w o r d 1 - - - | - - - w o r d 2 - - - |
| C 0 , C 1 , C 2 , C 3 | C 0 , C 1 , C 2 , C 3 | C 0 , C 1 , C 2 , C 3 |
| B 0 . . . . . B 3 1 | B 0 . . . . . B 3 1 | B 0 . . . . . B 3 1 |
```

Non-portable code which depends upon byte ordering for retrieving data must be rewritten.

4. ALL MEMORY REFERENCES ARE ALIGNED

Word and half-word data are stored most significant byte first and aligned on natural boundaries. Off-boundary storage references are not supported. The low two or one address bits are silently ignored, creating unexpected results.

If *lint*(1) is run against such programs, it complains about a "possible alignment problem."

5. FLOATING POINT IS IEEE STANDARD

IBM/4.3 conforms to IEEE Standard 754 for binary floating point arithmetic. The article "Floating Point Arithmetic" in Volume II notes the differences from VAX floating point.

A class of programming errors easily overlooked on the VAX -- treating the first half of a **double** quantity as a **float** quantity, or vice versa -- is highly visible on the RT PC. If numeric results are incorrect, look first for unions, casts, or function arguments that mismatch **double** and **float**. The *scanf* format "%f" instead of "%lf" is particularly subtle.

6. OLD CALLING SEQUENCE IS NO LONGER SUPPORTED

The subroutine calling sequence currently used in IBM/4.3 first appeared in the March 1986 release. As a transition aid, that release also supported the old calling sequence.

Beginning with the December 1986 release (PRPQ #5799-CGZ, Release 2), only the current calling sequence is supported. In the unlikely event that your installation still has programs not recompiled since you installed the March release, you must recompile and relink them. In the current release, running an old a.out will produce the message: **old calling sequence**, then terminate.

In the even more unlikely event that the source for the old program is no longer available, you can reinstate support for the old calling sequence in the current release (with a performance penalty) by specifying "option DUALCALL" in the kernel config file and rebuilding the kernel. See the article "Building IBM/4.3 Systems with Config" in Volume II.

Some of the IBM Support tools provided in the March release used the old calling sequence. Be sure to replace these by the versions provided in subsequent releases.

7. CAUTION WHEN USING THE 4.3 AT COMMAND

The 4.3 *at*(1) command does not pass the environmental variable **TERM** into a user's *at* spool file. Spool-file processing may break if the user's *.cshrc* file includes a reference of the form "$TERM" and the user's environmental shell is csh. To be defensive, csh users should code their *.cshrc* files in such a way as to test whether a variable is set before being referenced. For example:

```
if ($?TERM) then                    # is TERM defined?
        if ($TERM == h19) then
                setenv MORE -c
        endif
endif
```

(This is good programming practice for *.login* files as well).

8. CAUTION WHEN USING THE 4.3 CSH ON SETUID SCRIPTS

The 4.3 *csh*(1) command requires that a **-b** flag be used on the interpreter line of setuid *csh* scripts. *Csh* exits with a "Permission denied" error message if the **-b** flag is not specified.

This page intentionally left blank.

IBM/4.3 Linkage Convention

ABSTRACT

The IBM/4.3 linkage convention provides an efficient method of calling, executing and returning from functions. The convention provides support for customary facilities of C, FORTRAN, and Pascal, including *varargs*, *alloca*, and profiling.

This article is intended for compiler writers and others who must write or analyze programs at the machine-instruction level. It presumes understanding of the IBM RT PC or IBM 6152 Academic System architecture and the IBM/4.3 assembler language.

Also described is the Floating Point Arithmetic linkage, which presents a low-overhead, uniform interface to the various types of floating point hardware as well as a software emulator.

1. Introduction

A C function **foo** consists of a *text area* and a *data area*. The data area is named **_foo** and, in addition to quantities specified below, may contain constants and initialized variables. The text area contains machine instructions followed by a *trace table* that provides auxiliary information for debuggers.

Each call of **foo** creates a *stack frame* containing arguments, local variables, and space to save the caller's registers to be restored on return to the caller.

When **foo** is called, the caller first prepares an argument list, then transfers control to the text location named **_.foo**, which is **foo**'s entry point. A stack frame is built by **foo**'s *prolog* to hold local variables and saves any registers that are to be preserved for the caller. Execution proceeds through the body of **foo**, possibly calling other functions, and ends in the *epilog*, which prepares the return value, restores the caller's registers, releases the stack frame, and transfers control back to the caller.

2. Stack Usage and Stack Frame Format

The stack holds frames for currently active functions and signal handlers. The stack will occupy the highest possible locations in the core image, growing downward from 0x1fffe000. The stack is automatically extended as required. The data segment is only extended as requested by *brk*(2). A "red zone" of protected addresses separates the stack from the data segment, which starts at 0x10000000 and may grow upward as the result of *brk*(2) and *sbrk*(2) usage. Register r1 indicates the low address of the stack frame of the currently executing function. Locations above (r1) − 0x64 are preserved over interrupts. Locations below (r1) − 0x64 are considered unallocated storage and may be overwritten if a signal handler is activated. Figure 1 is a graphic representation of the stack frame format.

The stack is not self-describing, but with information from the trace tables in program text, a debugger can decompose the stack into frames and backtrace through it.

foo's stack frame holds the following areas, from lowest address (bottom of the figure) to highest (top of the figure):

a) Words 5 through *pmax* of outgoing argument lists. *pmax* represents the number of words in the longest argument list for functions that **foo** calls.

b) Local variables: autos and temporaries.

c) 0 or 18 (six registers * 12 bytes) or 64 (reserved for future expansion) words of save area for caller's floating point registers.

d) 1-16 words of save area for caller's general registers.

e) 1 word of static link for Pascal procedures: pointer to enclosing procedure's frame. Not used by C or FORTRAN.

f) 4 words of linkage area are reserved. Two words are now used to save floating point register 6.

g) 4 words allocated for the first four words of **foo**'s incoming argument list.

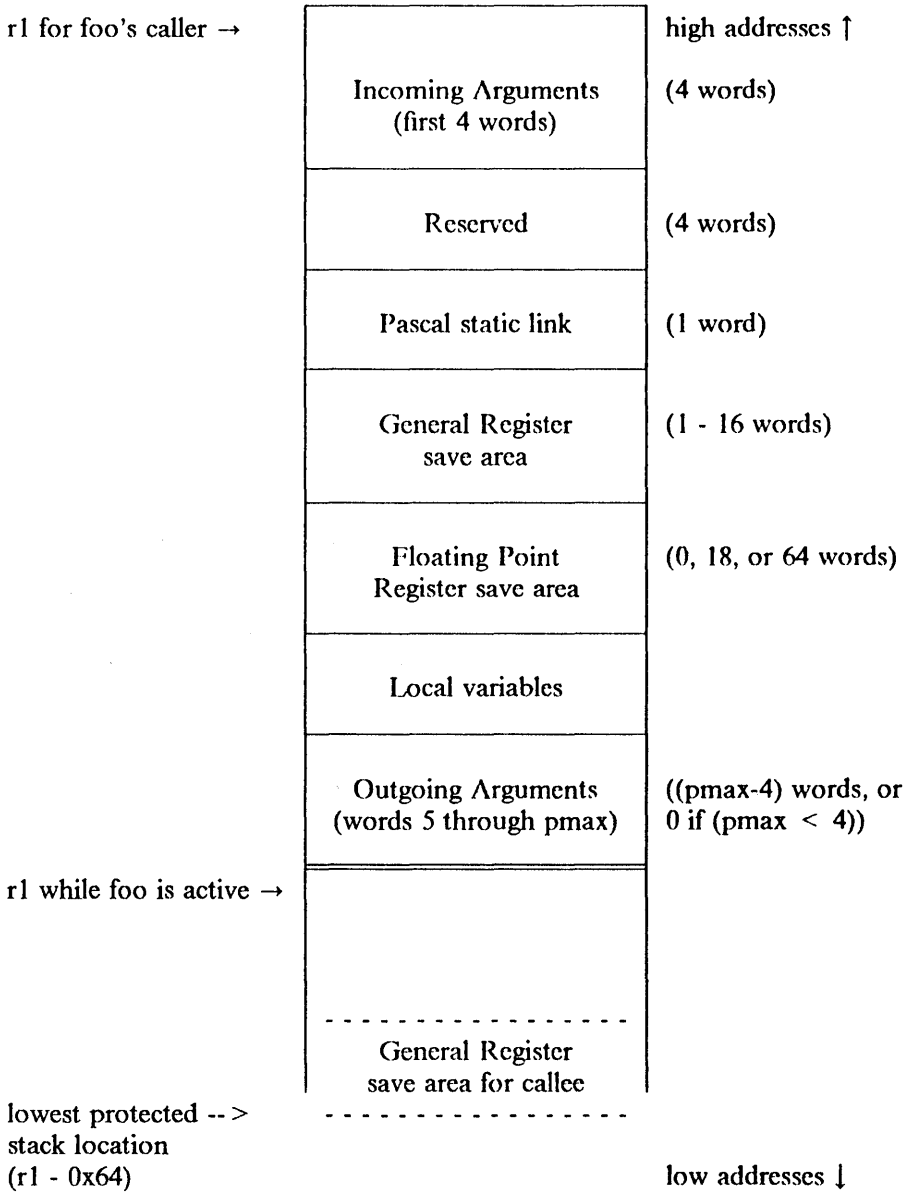


FIGURE 1. A STACK FRAME

foo can use the Store Multiple (stm) and Load Multiple (lm) instructions to save and restore registers, from any starting register through r15. However, registers r0-r5 do not need to be preserved. Prolog and epilog examples below show how the caller's r1 is restored.

The floating register save area holds up to 4 doubleword registers ending with register 5. No space is allocated if no floating registers need to be preserved.

The file */usr/include/frame.h* gives symbolic definitions for the sizes and offsets of some of these areas.

3. Register Usage

Certain registers, such as r1, have specific uses throughout execution; others, like r15, are specified during a call and are free at other times. The following table defines register usage at the call interface.

Register	Preserved over call	Usage
r0	no	called function's data area pointer
r1	yes	stack pointer (to caller's frame)
r2	no	argument word 1 and returned value
r3	no	argument word 2 and second word of a returned double value
r4	no	argument word 3
r5	no	argument word 4
r6-r13	yes	register variables, etc.
r14	yes	data area pointer (not required)
r15	no	return address
mq	no	multiply/divide register

r1 always addresses the bottom of the stack frame of the currently executing function. A compiler may assign another register to address the high end of the stack frame. The portable C compiler, for instance, points r13 at the last 64 bytes of auto storage. The linkage convention requires this second register only for *alloca* support (see the section entitled **Alloca Storage Allocation** below). The register number and the offset from the frame top, which are arbitrary, are recorded in the trace table.

Floating-point registers 0 and 1 are not preserved over a call. Registers 2 - 7 must be preserved. Floating point registers are not used to pass arguments or return results.

4. The Data Area

The data area (also called "constant pool," which is a misnomer) is addressed by r0 on entry to **foo**. The word pointed to by r0 must contain **_.foo**, the address of **foo**'s entry point. The following word supports the profiling option, and if present must be initialized to zero; the third word, also optional, supports *alloca* storage allocation.

It is conventional, but not required, for r14 to address the data area during execution. (The optional profiling linkage, which follows the prolog, does require it momentarily.)

For easy addressability, other data such as static variables, strings, or a literal pool may be located in the data area, either before or after the word addressed by r14.

A value **&foo** of type pointer to function corresponds to the address of **_foo**, the function's data area, not the address of **_.foo**, the function's entry point. A program that does arithmetic on function pointers, assuming that they address entry points, will probably malfunction.

5. Argument Lists

Arguments are word-aligned and allocated to consecutive stack locations. The list spans frame boundaries: words 1-4 are allocated in the top of the called function's frame, and the remainder are stored in the bottom of the caller's frame, which is adjacent. Argument words 1-4 are passed in registers r2-r5, not on the stack. The called function may choose to store them in the allocated stack locations, but this is not necessary except in a function like *printf* which accesses its argument list via a pointer variable. Such functions must use the *varargs*(3) macros to assure that argument words 1-4 get stored properly.

Arguments are passed as follows, based on argument type:

- An int is passed in a single word.
- A long, short, pointer, or char is treated as an int and passed in a word. A function pointer is represented by the address of the function's data area.
- A double is passed in two successive words.
- A float is converted to a double and passed in two successive words.
- A structure is word-aligned to a full word and left justified, except for structures of 1, 2, or 3 bytes, which are right justified.

If the function is declared to return a structure, the caller passes the address of a result area in r2, and word 1 of the explicit argument list is passed in r3. Subsequent argument words are shifted accordingly.

6. Calling Sequence

A typical call of a function **foo** first prepares the argument list, then executes the following:

```
balix   r15,_.foo          # call
l       r0,$.long(_foo)    # get its data area pointer
```

Dereferencing a function pointer calls a function without needing to know its name. Suppose that the function pointer, which addresses the function's data area, is in r8. A typical instruction sequence is:

```
ls      r7,0(r8)    # get address of entry point
balrx   r15,r7      # call whomever
mr      r0,r8       # r0 = data area pointer
```

7. Prolog

The prolog saves the caller's registers and obtains stack space for the stack frame. A typical instruction sequence is:

```
_.foo:  stm   r10,-60(r1)      # save caller's registers
        ai    r1,-framesize    # allocate our stack frame
        mr    r14,r0           # initialize data pointer
```

Other instruction sequences are needed for frame sizes larger than 32768 bytes. A sequence that decreases r1 in two stages is acceptable if the stack remains protected at all times. An example of an *un*acceptable sequence for a frame size of 64536 is

```
cal   r1,-bothalf(r1)   # -bothalf = 1000
cau   r1,-tophalf(r1)   # -tophalf = -1
```

This momentarily increases r1, letting an ill-timed interrupt destroy the stack.

8. Profiling

If either the -p or -pg option is selected, this instruction sequence follows the prolog and accomplishes data collection for performance monitoring:

```
mr    r0,r15
bali  r15,mcount   # r0  = caller's return address
                   # r14 = our data address
                   # r15 = our return address
```

9. Epilog

The epilog prepares a result, restores the caller's environment, and returns control. A typical instruction sequence is:

```
lis   r2,0                  # zero result returned in r2
lm    r10,framesize-60(r1)  # restore registers
brx   r15                   # return
ai    r1,framesize          # adjust stack frame
```

The location of the return value depends on the function type:

- An int, long, short, pointer, or char is returned in r2.
- A double is returned in r2 and r3.
- A float is returned as a double.
- A structure result is returned by moving it into the area pointed to by the first argument list word (in r2 on entry).

10. Alloca Storage Allocation

The implementation-dependent storage allocator *alloca* (see *malloc*(3)) expands its caller's stack frame by decreasing r1, to obtain a storage area that is automatically deallocated on return. The storage area so obtained starts at the end of the maximum-length argument list in the newly expanded frame. *alloca* can be called from any function that follows two conventions:

(1) It addresses outgoing argument lists through r1, and addresses all other areas in the stack frame through some other register (identified in the trace table as *frame_reg*).

(2) In its data area, which must be addressed by r14, the halfword at (r14)+8 holds the value 0xf690 (a magic number, used for validity checking). The halfword at (r14)+10 holds the length of the longest outgoing argument list (exclusive of the first four words, which do not occupy space in the frame).

Files compiled with the *hc*(1) or *pcc*(1) option **-ma** adhere to these conventions.

11. Trace Tables

Debuggers rely on a trace table of 6-10 bytes following the text of each function. A debugger locates a trace table by searching forward through program text (generally from a point indicated by a call's return address). The search stops when it finds two successive halfwords, each having 0xdf in its first byte. For compiled C functions, or assembler functions following the same conventions, the trace table corresponds to the following structure:

```
struct TT_D_COM {
unsigned magic1 : 8          = 0xdf,
         code   : 8          = 7,
         magic2: 8           = 0xdf,
```

```
        first_gpr : 4,
        optw  : 1                   = 1,
        optx  : 1,
        opty  : 1,
              : 1                   = 0;
char    npars : 4,
        frame_reg : 4;
char    fpr_save  : 8;              /*this byte present only if opty = = 1*/
char    lcl_off_size : 2,           /* lcl_offset is variable length */
        lcl_offset1 : 6,
        lcl_offsetn[lcl_off_size];
        }
```

first_gpr is the first register saved by the store multiple instruction in the prolog. This indicates the size of the general register save area.

opty is 1 if the byte holding *fpr_save* is present; otherwise, it is 0.

npars is the number of words of declared arguments. The maximum value of 15 does not restrict the actual length of argument lists.

frame_reg identifies the register used to address local variables, etc., in the stack frame. *frame_reg* is 1 unless an alternative register is used (for example, r13 for pcc-compiled functions).

fpr_save is present only if *opty* is 1. It is a 6-bit mask (right-justified) indicating which floating point registers are saved. The rightmost bit corresponds to floating point register 7, thus:

0	0	x	x	x	x	x	x
-	-	-	-	-	-	-	-
		fr2	fr3	fr4	fr5	fr6	fr7

If *fpr_save* is 0 or not present, no floating registers are saved.

lcl_offset is an unsigned integer 6, 14, 22, or 30 bits long. It indicates the distance, in words, to the top of the stack frame from the point addressed by register *frame_reg*.

12. as(1) Routines

A very simple C function or hand-coded assembler function, that doesn't call other functions, can take some shortcuts. It may not need to save and restore registers, and need not allocate a stack frame if the protected area between (r1) − 0x64 and (r1) gives it sufficient storage.

Such a function has a simpler trace table: the four byte sequence 0xdf02df00. Debuggers may not be able to backtrace from this function if the caller's r14 and r15 are disturbed.

Temporarily each file must include the lines:

```
.globl   .oVncs
.set     .oVncs,0
```

This is used by *ld*(1) to detect use of an obsolete linkage convention. Compilers generate definitions of .oVncs automatically.

13. Addressability in Very Large Modules

When .o files are linked by *ld*(1), the resulting object module may be so large that the text of the caller and callee are more than 2^{20} bytes apart. The balix instruction in the call cannot

then address the callee, and *ld* modifies the instruction in one of two ways to establish addressability.

A balax replaces the balix if it can duplicate the balix's effects, that is, the callee's address is below 2^{24} and r15 is the link register. Otherwise, the balix is replaced by a balix to a piece of "trampoline code" that derives the callee's entry point address from the contents of r0 and branches to it.

Other than in function calls, addresses are always carried as 32-bit values, so addressability is unaffected by module size.

14. Floating Point Arithmetic Linkage

IBM/4.3 provides a floating point environment in which executable programs will run on all supported models of the IBM RT PC, selecting at runtime the highest performance floating point hardware installed in a given machine.

14.1. What the Compiler Must Do

The key to this environment is that the compiler(s) produces generic floating point instructions rather than instructions specific to any particular floating point hardware. These generic instructions are generated as follows:

(1) The compiler sets aside a small block in the data area.

(2) In that block the compiler places the following:

  (a) an instruction to save general register 15 (the return address)

  (b) an instruction to branch to a pre-defined location, the "glue code."

  (c) a description of the specific floating point instruction (such as add)

  (d) the operands for the instruction (such as one or more floating point registers)

  (e) some additional information

  This information follows the well-defined format described below, but is not specific to any floating point hardware.

(3) At the point in the code where the floating point instruction is to be performed, the compiler generates a branch-and-link instruction to the block it set up in the data area. Note that if at some point later in the code generation the compiler wants to generate the same floating point instruction (such as to add the same two floating point registers), it need only place a branch-and-link instruction to the same block in the data area.

When finished, the compiler will have generated branch-and-links in all places where floating point instructions occur, and a block in the data area for each unique instruction.

At execution, via a system call, the glue code gets the address of a floating point "code generator" in the kernel and invokes it. This code generator interprets the generic instruction stored in the data area and translates it into code appropriate to the floating point hardware installed or the emulator.

14.2. Compiler Specifications

The generic instructions are intended to hide floating point hardware specifics from both the user and the compiler-writer. There are, however, important aspects of floating point which, though ultimately dependent on particular hardware, can be guaranteed by IBM/4.3 and, indeed, must be known by the compiler. A discussion of these aspects follows.

14.2.1. IEEE Floating Point

The most important of these aspects is that floating point logic adheres to IEEE Standard 754. The compiler writer can assume that all floating point performed by this support will conform to the standard for single and double precision. Likewise, this support assumes that the compiler will adhere to the standard in those cases which are affected by the language (such as four-way compares).

14.2.2. Floating Point Registers

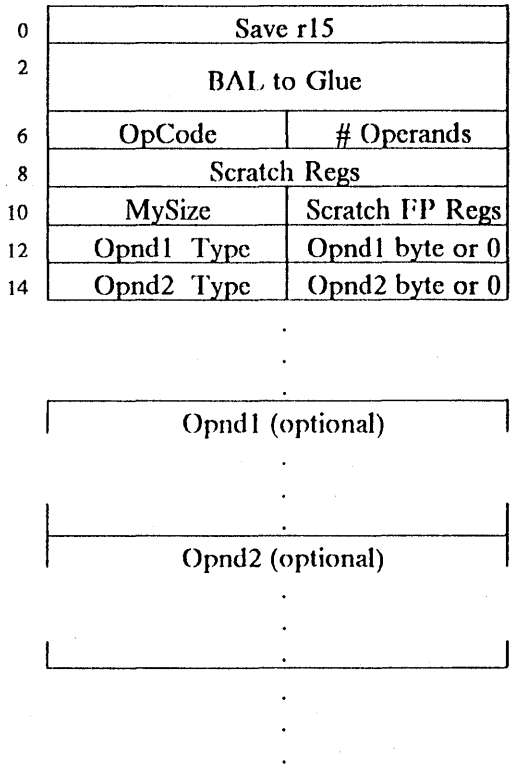
Floating point registers are referred to as fr0,fr1,...; general registers are referred to as r0,r1,....

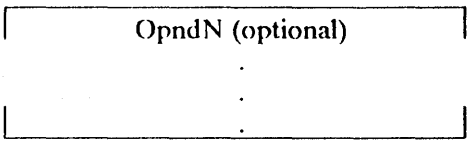
IBM/4.3 Floating Point support assumes there are eight floating point registers available (fr0 through fr7). Each register can contain either a single or a double precision value. The compiler will be responsible, if it so chooses, for managing these registers (for intermediate results, or register variables).

The registers fr0 and fr1 are considered scratch registers. Registers fr2 through fr7 must be saved across function calls; load- and store-multiple instructions for these registers are provided. A save area of 6*12 bytes (18 words) must be reserved in the stack frame if any floating point registers are saved (not 8*(# registers saved) as in 4.2/RT).

14.2.3. Format for Generic Instructions

The following illustration graphically depicts a general data area block for a floating point instruction, as a variable-length array of half-words. The start of each block must be full-word aligned, and certain operands may require full-word alignment--see description of operands below.





The first instruction, **Save r15** moves r15 into r0: "cas r0,r15,r0".

The second instruction, **BAL to Glue** will be ".long FPGLUE", where FPGLUE is defined in libc with ".set FPGLUE,fpglue + 0x8a000000" (a bala to fpglue, the actual glue code).

OpCode is a byte specifying the particular operation being done. The opcodes for each operation are defined in *< machine/rtflops.h >*.

The next byte, **# Operands**, specifies the number of operands for this instruction. If the operation requires *n* operands, this number may be either *n* or *(n+ 1)*. The result of the operation is always placed in operand 1, and the operation is performed on the last *n* operands. In summary,

0

Floating Point Instruction	Type of Operation	Description
Op x	Monadic	x <- Op(x)
Op x y	Monadic (or move)	x <- Op(y)
Op x y	Dyadic	x <- Op(x,y)
Op x y z	Dyadic	x <- Op(y,z)

Scratch Regs is a bit map of the general registers available for the code generator to use (the scratch registers at this point in the program). Bits 0-15, numbered left to right, each represent a corresponding general register r0-15. If a particular bit is set, that general register is available for use. However, r0 and r15 are always considered scratch; their bits will be ignored, but should always be set. The multiplier-quotient (mq) register is also considered scratch. General register operands which do not need to be preserved over the operation can and should be marked scratch. **NOTE:** If the code generator needs more registers than are available, it will use the stack to save and restore register contents. For this reason, the compiler must insure that the stack below the stack pointer (r1) is available; floating point instructions must be generated only *after* r1 has been decreased to allocate the stack frame in prologs and *before* the caller's r1 has been restored in epilogs.

MySize is a byte containing the size of each floating point instruction block, measured in bytes. (Since the size and number of operands varies, the compiler must tell the code generator how big the block is.) A minimum of 12 bytes is required. See Appendix A of this article for the rules dictating the size of a floating point instruction block.

Scratch FP Regs is similar to **Scratch Regs**, but is one byte long, and each bit corresponds to a floating point register fr0-fr7. Floating point register operands which do not need to be preserved over the operation can (and should) be marked scratch. Specifying any scratch floating point registers is not required, but may help performance in the following cases:

(1) Moves and monadic operations where no operands are floating point registers (one scratch register will help).

(2) Dyadic operations where only one operand is a floating point register (one scratch register will help).

(3) Dyadic operations where no operands are floating point registers (two scratch registers will help).

OpndN Type is a byte specifying the type of its corresponding operand. The byte immediately following the type is the operand itself (or a part thereof) for register-number operands, or it is 0. The one (or two) word values of operands follow all of the types and byte operands (in their respective order) to allow easy full-word alignments. The following table lists operand types; all are defined in *< machine/rtflops.h >*.

OPERAND TYPES

Operand Size	Operand Value
1 byte	General register number, containing a signed or unsigned integer or single precision
1 byte	Two general register numbers, containing a double precision (high half in register specified in high-order nibble; low half in register specified in low-order nibble)
1 byte	Floating point register containing a single or double precision
1 word	Immediate value, signed or unsigned integer, or single or double precision
2 words	Immediate value, double precision
1 byte + 1 word	Address of a signed or unsigned integer, a single or double precision, or a function pointer (a general register number + a 32-bit offset)
varies	Special value, dependent on special OpCode

Some operand types may have no meaning in certain circumstances. For example, an immediate value cannot be the result of an instruction. All full-word operand values (immediate and address-offset) must be full-word aligned.

14.2.4. Operations and Operands

Most of the operations fall into the monadic or dyadic category and are treated similarly. All of the integer and floating point operand types shown in the table above are legal operand types for these operations. The same types, with the exception of the immediate values, are also valid results. All of the monadic and dyadic operations, except CMP and CMPT, allow an extra operand; this means the first "operand" is the result, and the following operands are the actual arguments to the operation.

Comparisons (opcodes CMP and CMPT) require two operands, and set the RT condition code appropriately (GT, EQ, LT, or none--the latter if the comparison involved a NaN). CMP generates code for a non-trapping compare, and should be used on comparisons of equality (and inequality) only. CMPT generates code for a trapping compare, and should be used on all comparisons of order (LT, GT, LE, and GE). (CMPT will cause an exception when comparing NaNs.)

Operations on operands of differing precision will always involve the appropriate conversions (converting to wider precision, performing the operation, and converting back to destination precision). A MOVE from one precision to another implies a conversion; MOVEs from floating point to integers will always truncate.

The STOREM and LOADM operations (store-multiple and load-multiple) take two operands, both of type "special." The first operand is a bit-map (stored in the byte following the operand type in the instruction) marking the floating point registers to be saved or restored. The second operand is the address of the save area, in the same register/offset format as other address-type operands.

14.3. User-Level Interface

This section describes the glue code that accommodates IBM/4.3 floating point support.

The code generator, like the floating point emulator, will be linked into the kernel. This has a significant advantage over linking it directly into user programs, as subsequent hardware releases will require only a new kernel; existing programs do not have to be recompiled *or* re-linked. The source for the code generator is in */usr/src/usr.lib/lipfp/genfp*.

The glue code is linked into crt0.o so that it will always be addressable by a "bala." (The source for the glue code is in */usr/src/lib/libc/machine/csu/fpglue.s*.)

The glue code, when executed the first time, will set up appropriate information needed for code generation. (Most of this information will be gotten from the kernel via system calls.) Then it will invoke the code generator. Subsequent "calls" to the glue code will only invoke the code generator.

14.4. Compatibility with Previous Releases

This initial release of the new IBM/4.3 Floating Point support gives the maximum compatibility possible with previously linked programs. (Note, however, that future releases may not provide this compatibility.) Most a.outs compiled and linked under previous releases will produce the same results when run under this release. However, performance will be improved by recompiling, especially if running on an RT with an APC. See further notes in the article entitled "Floating Point Arithmetic."

REFERENCES

(1) Johnson, S. C. and D. M. Ritchie. "The C Language Calling Sequence," Computing Science Technical Report No. 102, Bell Laboratories, Murray Hill, NJ, 1981.

(2) "Assembler Reference Manual for IBM/4.3," in Volume II, Supplementary Documents.

(3) "Floating Point Arithmetic," in Volume II, Supplementary Documents.

(4) *IBM RT PC Hardware Technical Reference*, SV21-8024

APPENDIX A. RULES GOVERNING INSTRUCTION BLOCK SIZES

A minimum of 12 bytes is required for each floating point instruction. If more space is required, routines are included to calculate and allocate such space. However, this may degrade performance. Another alternative is to allocate a maximum size, but this is an inefficient use of space. To fine tune a program that uses floating point arithmetic, consider using the following rules to determine the size of the data block required for an instruction.

Most operations:
24 MINIMUM

Operands:
+ 32 each non-floating point register operand, or fr7 (because fr7 is not on the FPA, it should be considered a NON-floating point register)
+ 12 for each conversion
+ 12 if 3rd operand type != 1st operand type

Other:
+ 32 if operation is CMP
+ 64 if operation is CMPT

Loadm, Storem:
34 MINIMUM (one register being saved/restored)

+ 24 for each additional register
+ 8 if one of the registers is fr7

General scratch registers:
+ 16 if no scratch
+ 8 if 1 scratch
+ 4 if 2 scratch
+ 0 if 3 or more scratch

Floating point scratch registers:
Monadic operations:
IF 1st operand is a floating point register, no scratch needed.

IF 1st operand is not a floating point register:
+ 64 if no scratch
+ 0 if 1 or more scratch

Dyadic operations:
IF no operands are floating point registers:
+ 112 if no scratch
+ 64 if 1 scratch
+ 0 if 2 or more scratch

IF 2 operands:
IF both floating point registers, no scratch needed.

IF 1st operand is a floating point register AND the 2nd operand is not:
+ 64 if no scratch

+0 if 1 or more scratch

IF 3 operands:

IF all floating point registers, no scratch needed.

IF 1st and 3rd operands are different floating point registers,
no scratch needed. If they are the same, follow next rule.

IF 1st operand is a floating point register AND the 3rd operand is not:
+64 if no scratch
+0 if 1 or more scratch

IF 1st operand is NOT a floating point register AND the 3rd operand IS:
+64 if no scratch
+0 if 1 or more scratch

IF 1st and 3rd operands are NOT floating point registers:
+112 if no scratch
+64 if 1 scratch
+0 if 2 or more scratch

Recompiling with High C

ABSTRACT

Both *pcc* (the standard C compiler provided with Berkeley systems) and MetaWare High C are available in IBM/4.3. High C offers significant advantages over its predecessor and is the default C compiler. This article serves as a guide for C programmers in recompiling existing programs with High C. The article contains three chapters:

1. **Introduction** describes High C, contrasting it with *pcc*.
2. **Diagnostic Messages** explains a sample High C diagnostic message and describes messages frequently encountered when recompiling programs with High C.
3. **Run-Time Differences** describes those differences between *pcc* and High C that may not manifest themselves until run-time.

1. INTRODUCTION

IBM/4.3 now provides a new optimizing C compiler, MetaWare High C, in addition to the standard *pcc*-based C compiler. High C provides extensive code optimization, producing compiled programs that run up to twice as fast as *pcc*-compiled programs. It also generates tighter code; object file text is typically 15% smaller than with *pcc*.

Hc has been tested against the C Test Suite provided by Human Computing Resources Corporation, and is used to compile the entire IBM/4.3 system (with the exception of assembler routines and a few other files).

The commands *hc*(1) and *pcc*(1) are available in the */bin* directory. Users are not obliged to use one compiler or the other. The command *cc*(1) in */bin* is a symbolic link that may point to either *hc* or *pcc*. In the IBM/4.3 system as distributed, */bin/cc* points to *hc*.

The *hc* feature you will notice first is probably its meticulous semantic and syntactic checking and precise diagnostics. Many old programs that compile "error free" with *pcc* generate warnings and errors with *hc*, usually for good reason. In recompiling IBM/4.3, we found that messages sometimes pointed out type mismatches, incorrect-length argument lists, and uninitialized or misspelled variables that had been undetected for years. The "High C Programmer's Guide" tells how to use flags and toggles to adjust the error and warning sensitivity up or down; we recommend "up" during program development.

High C represents a significant step toward the draft ANSI C standard, and supports a more extended C language than does *pcc*. The *High C Language Reference Manual* describes the extensions in full. One extension that may affect existing programs is the presence of new keywords: **signed**, **const**, and **volatile** for ANSI, plus **pragma** (borrowed from Ada). A program using any of these four names for identifiers will have to be modified, for instance by adding the line:

```
# define const _const
```

Two other ANSI-related changes, character escapes and widening rules, are discussed in the sections on "Character Escapes" and "Integer Widening" below.

In general, High C supports the semantics of "classical" C, where this is not precluded by adherence to the draft ANSI C standard. Even so, there are circumstances in which a language construct that is incompletely defined may execute differently when compiled with *hc* and *pcc*. Chapter 3, "Run-Time Differences," discusses constructs whose semantics may differ.

2. DIAGNOSTIC MESSAGES

This section provides an explanation of a sample diagnostic message and includes a list of diagnostics frequently produced when recompiling with *hc*. The list provides an explanation of each diagnostic and, where appropriate, a recommended solution.

2.1. Sample Diagnostic Message

The following shows a code fragment, a diagnostic message generated by the code, and an explanation of the message.

Code Fragment:

```
1       /* this file is named test.c */
2
3       main()
4
5       {
6        char *j;
7        int  i;
8
9        i = j + i;
10
11      }
```

Diagnostic Message:

```
E "test.c", L9/C5:
|     Type *Unsigned-Char (at "test.c", L6/C6) is not assignment compatible with type Signed-Int.
```

Explanation:

- The "E" stands for Error. Warning messages begin with a "w."
- "test.c" is the name of the module containing the error.
- L9/C5 indicates the error was detected in Line 9, Column 5.
- The body of the error message explains that a value (j + i) of type pointer to **unsigned char** was being assigned to a variable (i) of type **signed int**. This is illegal (but unchecked by *pcc*).
- The phrase (at "test.c", L6/C6) locates the declaration that gave rise to the value of type pointer to **unsigned char**. This is particularly helpful in locating declarations in #include files.
- The vertical bar "|" in the first column indicates a continuation line of a multiline message.

2.2. Common Diagnostic Messages

This section lists the most frequently encountered messages and suggests ways to resolve them. See the section on diagnostic messages in the "High C Programmer's Guide" for a complete list of warning and error messages.

Type t is not assignment compatible with type t'.

The mismatched-type message appears for any of several reasons. Most frequently, it has to do with pointer conversion, and can be eliminated by using explicit casts. In this example, the comments propose ways to rewrite each statement.

```
main()
{
        char *pc;
        int *pi, i, x;

        pc = pi;        /* should be: pc = (char *)pi; */
        x = pc + i;     /* should be: x = (int)(pc + i); */
        i = pc;         /* should be: i = (int)pc; */
}
```

Another common cause of this message is shortcuts in structure initialization. As an example, given the declaration:

```
struct s1 { int i, j; };
```

the shortcut initialization:

```
struct s1 x = 0;
```

is allowed by *pcc*, but C syntax (and *hc*) require braces around the initializer:

```
struct s1 x = {0};
```

Variable is set but is never referenced.

This message warns of an initialized variable that is not used in the module. It may be a symptom of a logic error.

This diagnostic prints in another common situation: if RCS or SCCS variables are contained in the program header. In this case, you can ignore the message.

Result of comparison never varies.

An expression was found whose operands are such that the value of the expression is always the same. The usual cause is a logic error arising from confusion over signed/unsigned types. For example, an **unsigned char** cannot be negative; therefore, a comparison with a negative constant will never vary. Look for assumptions that the type **char** is really signed.

Variable required.

This generally points out an illegal left-hand side of an assignment. This error can be produced by statements of the form:

```
(CONDITION ? i : j) = -1;
```

which *pcc* (incorrectly) allows if CONDITION involves only constants and preprocessor variables. Rewrite it as:

```
*(CONDITION ? &i : &j) = -1;
```

This is multiply-declared.

This may be the result of a variable declared **extern**, then redeclared later in the same module as **static**. This is often caused by an **extern** declaration in an #include file. *Pcc* allowed the redeclaration. Correct this by using distinct names for the two variables.

Local function is never < referenced; no code will be generated for it.

A function of storage class **static** is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function, and it is eliminated as dead code. The -g option disables this optimization, so that *dbx*(1) sessions can access such functions.

Expression has no side effect and has been deleted.

The value of an expression is not assigned to a variable or otherwise used to affect the computation. For example, "2+3;" is useless and is deleted.

This function declaration is inconsistent with the "int"-returning function declaration imputed at Ln/Cm.

A function that is called before it is declared is assumed to return **int**. Any subsequent declaration of the function must declare it to do so.

Correct this by placing an explicit declaration of the function with the proper return type before the first call (and check all calls for their assumptions about the return type!).

Unexpected char.

Pcc allows multi-character character constants; *hc* does not. For example, for the following declaration:

```
int x = 'abcd';
```

pcc assigns the value 0x61626364 to *x*, but *hc* generates the above error message.

Fewer arguments given than function has parameters.

Hc checks argument lists in calls of functions that are declared in the same module.

3. RUN-TIME DIFFERENCES

Some of the differences between *hc* and *pcc* will not manifest themselves until load- or run-time. This chapter describes these differences and provides an explanation for their causes.

3.1. Order of Execution

C semantics permit subexpressions in a larger expression to be evaluated in any order, or even concurrently. The statements

```
i = j + j+ + ;
foo( i, i--);
```

do not have well-defined meanings and may well execute differently with *hc* and *pcc*. To assure that side effects like assignment occur in a defined sequence, break such expressions into multiple statements.

3.2. Multiple Assignments

Look out for multiple assignments that require both narrowing and widening integer values, such as:

```
int i; char c;
i = c = integer-expression;
```

Here the integer-expression is "narrowed" on assignment to c. Language rules require (and *hc* supports) assignment of the narrowed value to i, not the original value. Code generated by *pcc* often fails to narrow the value correctly, and some incorrect programs may execute as intended only because of this *pcc* bug. Reorder the assignments, or write two statements.

3.3. Keyword "asm" Not Supported

Pcc allows inclusion of assembler statements within C programs via the "asm" construct. As *hc* does not produce intermediate code and generates code which is optimized across statements, this keyword is not supported.

Existing code which contains "asm"s will generate errors at load-time, with "_.asm" and "_asm" as unresolved references.

3.4. Volatile Memory

Hc optimizes the following code:

```
if (*p = = 0) buf = *p;
```

by loading the contents of location p into a register for the comparison, then using this same register for the assignment as well. If p is the address of memory that is volatile (for instance, it is an I/O register that is updated after each reference), the assignment will not reflect the changed value. Correct the problem by declaring p **volatile**. Or, since this type of code is common in device drivers (and other portions of the kernel), use the flag (-**Hvolatile**) to disable all common subexpression recognition across statement boundaries.

3.5. Use of *setjmp*(3) and *longjmp*(3)

ANSI specifies the values of local variables changed between a call of setjmp() and of longjmp() to be indeterminate after longjmp is called. Despite this, most implementations reliably update auto variables, and many existing programs rely implicitly on auto variables having current values after a longjmp. (Register variables are chancier.)

To accommodate this practice, *hc* recognizes the use of the function names "setjmp" or etjmp" to assure that auto variables are reliably updated.

3.6. Character Escapes

Hc supports the draft ANSI complement of character escapes:

\a alert (bell)	\t horizontal tab
\b backspace	\v vertical tab
\f form feed	\x*nnn* hexadecimal numeric
\n newline	\' single quote
\r return	\" double quote

Use of an undefined character escape results in a warning message.

3.7. Integer Widening: Value-Preserving vs. Unsignedness-Preserving

Historically, C compilers have used either of two widening rules: unsignedness-preserving (u-p) widens an unsigned **char** or **short** to **unsigned int**; value-preserving (v-p) widens it to a **signed int**. U-p is sometimes useful but creates many anomalous situations. Note the following example.

```
void f ()
{
    unsigned char c = getchar ();

    if (c - '0' < 0 || c - '0' > 9)
       printf("This character is not a digit");
}
```

Because *pcc* uses the u-p rule, the test (c - '0' < 0) will always fail (since an **unsigned int** can never have a value less than 0). Because *hc* uses the v-p rule, c will be widened to a signed integer; the test will work as expected. The v-p rule almost always produces the expected result, and is the rule chosen by the ANSI committee in the draft standard.

References

- Appendix C of this manual, which contains the "High C Programmer's Guide"
- *High C Language Reference Manual*, available from:

  MetaWare Incorporated
  903 Pacific Avenue, Suite 201
  Santa Cruz, CA 95060
  (408) 429-META

- Draft proposed American National Standard for the C Language; contact ANSI Committee X3J11 for the most recent draft.

This page intentionally left blank.

Professional Pascal Differences

ABSTRACT

Professional Pascal is available as a separately-licensed feature of IBM/4.3. Professional Pascal offers significant advantages over other Pascal compilers. It is a highly optimizing Pascal compiler that conforms to the ANSI Standard, and includes many useful extensions, such as support of varying length strings, bitwise operations, packages, and iterators.

This article points out the major differences between Berkeley Pascal and Professional Pascal, as an aid to programmers recompiling existing programs with Professional Pascal. The article has two chapters:

1. **Introduction** describes Professional Pascal, contrasting it with Berkeley Pascal.
2. **Significant Differences** briefly describes those differences that may prevent a program that compiles with Berkeley Pascal from compiling (or executing correctly) with Professional Pascal.

1. Introduction

Pascal programs which are not dependent upon a particular compiler's extensions, that is, programs written in ANSI Standard Pascal, should port to IBM/4.3 using Professional Pascal[1] with little or not effort. However, programs written in Berkeley Pascal may not port so easily. Berkeley Pascal includes many extensions to standard Pascal (which are outlined in Appendix A, "Appendix to Wirth's Pascal Report," of the "Berkeley Pascal User's Manual" in Volume 1 of *UNIX Programmers' Supplementary Documents*. If a program uses any of these extensions, it may not compile or execute as expected.

This article concentrates on the features of the Berkeley Pascal compiler[2] that are missing or differ from Professional Pascal. The article does not attempt to point out the many features of Professional Pascal which are not found in Berkeley Pascal. For a complete description of Professional Pascal extensions, please see *Professional Pascal Language Extensions Manual with Rationale and Tutorials*, available from MetaWare Incorporated.

2. Significant Differences

Several differences exist between *pp* and *pc* which may affect your programs. This section points out these differences.

2.1. Case of Identifiers

In *pc*, the names of identifiers are case-sensitive. In *pp*, they are case-insensitive; *all names* are shifted to lower case. Be sure all identifiers are uniquely named regardless of case.

2.2. In-Line Compiler Directives

Pc supports in-line control of compile-time options from within comments:

{$option}

Pp provides similar support of "toggle" setting via pragma statements. See "Compiler Toggles" in the *Professional Pascal Programmer's Guide*.

2.3. Octal Constants:

In *pc*, an integer constant is expressed in octal by a series of digits terminated with "B" or "b" (e.g. 777b). *Pp* precedes the digit series with the character string "8#" (e.g. 8#777).

2.4. New Reserved Words

In addition to standard Pascal keywords, the words **pragma**, **package**, **iterator**, **value**, and **otherwise** are reserved in *pp*. (They are not reserved in *pc*.)

2.5. Predefined Routines

The following predefined routines found in *pc* are not supported in *pp*:

- Predefined procedures: *date*, *flush*, *linelimit*, *message*, *null*, *pack*, *remove*, *stlimit*, *time*, and *unpack*.
- Predefined functions: *card*, *expo*, *random*, *seed*, *sysclock*, *undefined*, and *wallclock*.

---

[1] Hereinafter referred to as *pp*.

[2] Hereinafter referred to as *pc*. Note that what is true for *pc* in this article is also true for *pi*, the Berkeley Pascal interpreter. Therefore, *pc* can be taken to mean "*pc* and *pi*."

The routines *argc* and *argv* are not predefined as they are in *pc*, but they are defined in the "arg" package provided with *pp*. Note, however, the slightly different semantics for these routines as they are defined in "Utility Packages" in the *Professional Pascal Language Extensions* manual.

Similarly, the *clock* function is not predefined but is included in the *pp* "system" package.

The procedure *halt* is predefined in *pp* (as it is in *pc*), but it does not produce a control flow backtrace upon termination.

2.6. Writing Expression in Octal or Hexadecimal

In *pc*, the value i is displayed in octal by:

```
write(i oct)
```

or in hexadecimal by:

```
write(i hex)
```

where i is a **boolean**, **char**, **integer**, pointer or enumerated type. In *pp*, the equivalent would be:

```
write(ord(i):n:8)
```

or:

```
write(ord(i):n:16)
```

where "n" is the minimum field width.

2.7. Reading and Writing Enumerated Types

Reading and writing of enumerated types is not allowed in *pp*.

2.8. Associating File Name and Variable Name

In *pc*, a global file variable appearing in the **program** header is associated with a physical file of the same name. In *pp*, file variables appearing in the **program** header are associated with file names appearing as command-line arguments. See "Invoking the Compiler" in the *Professional Pascal Programmer's Guide*.

2.9. No Assert Statement

The **assert** statement of *pc* is not supported in *pp*.

2.10. Relational Operators on Sets

The relational operators "<" and ">" may not be applied to sets in *pp* as can be done in *pc*.

2.11. Simple Types Integer and Real

In *pc*, an **integer** is 32 bits wide. That is, it follows the conceptual definition:

```
type integer = -2147483648..2147483647;
```

In *pp*, an **integer** is 16 bits wide; it follows the conceptual definition:

```
type integer = -32768..32767;
```

Pp predefines the type **longint** to represent 32-bit integers; it is equivalent to *pc*'s type **integer**.

Pc represents a **real** in double-precision, or 64 bits. *Pp* represents **real** in single-precision, or 32 bits. *Pp* predefines the type **longreal** to represent double-precision; it is equivalent to *pc*'s type **real**.

If a *pc* program which is dependent on 32-bit **integers** and double-precision **reals** is ported to *pp*, the following redefinitions can be used:

```
type
    integer = longint;
    real = longreal;
const
    maxint = maxlong;
```

2.12. Predefined Types

Pc predefines the types **alfa** and **intset** as:

```
type
    alfa = packed array [1..10] of char;
    intset = set of 0..127;
```

These types are *not* predefined in *pp*; the above definitions can be added to existing programs that depend upon these types.

2.13. Subrange Mapping

In *pc*, the subrange 0..255 is mapped to a 16-bit word. In *pp*, it is mapped to an unsigned byte.

Pc maps the subrange 0..65535 to a 32-bit longword; *pp* maps it to an unsigned (16-bit) word.

2.14. Global Variables

In *pc*, all variables at the outermost level are made global static. In *pp*, such variables are made local static by default. The preferred way to share variables across modules in *pp* is via interface packages; however, the statement "pragma data(COMMON);" can be specified before the first variable declaration to achieve the same effect from *pp*.

2.15. Predefined Constants

Pc predefines the integer constant "minint"; *pp* does not. The following definition can be used:

```
const
    minint = -maxint - 1;
```

The predefined character constants "minchar," "maxchar," "bell," and "tab" of *pc* are not supported in *pp*.

References

- Appendix C of this manual, which contains the "High C Programmer's Guide"
- *Professional Pascal Documentation Set*, available from:

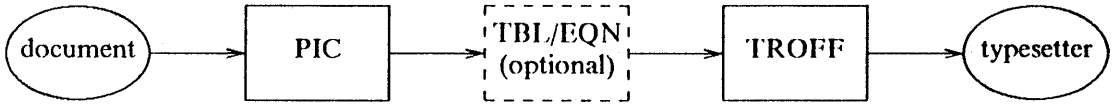
  MetaWare Incorporated
  903 Pacific Avenue, Suite 201
  Santa Cruz, CA 95060
  (408) 429-META

PIC – A Graphics Language for Typesetting
User Manual

Brian W. Kernighan

ABSTRACT

PIC is a language for drawing simple figures on a typesetter. The basic objects in **PIC** are boxes, circles, ellipses, lines, arrows, arcs, spline curves, and text. These may be placed anywhere, at positions specified absolutely or in terms of previous objects. The example below illustrates the general capabilities of the language.



This picture was created with the input

```
ellipse "document"
arrow
box "PIC"
arrow
box "TBL/EQN" "(optional)" dashed
arrow
box "TROFF"
arrow
ellipse "typesetter"
```

PIC is another **TROFF** processor; it passes most of its input through untouched, but translates commands between **.PS** and **.PE** into **TROFF** commands that draw the pictures.

1. Introduction

PIC is a language for drawing simple pictures. It operates as yet another **TROFF**[1] preprocessor, (in the same style as **EQN**[2], **TBL**[3] and **REFER**[4]), with pictures marked by **.PS** and **.PE**.

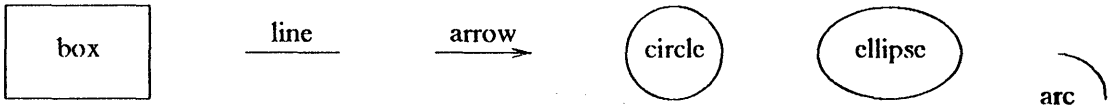
PIC was inspired partly by Chris Van Wyk's early work on **IDEAL**[5]; it has somewhat the same capabilities, but quite a different flavor. In particular, **PIC** is much more procedural – a picture is drawn by specifying (sometimes in painful detail) the motions that one goes through to draw it. Other direct influences include the **PICTURE** language [6] and the V viewgraph language [7].

This paper is primarily a user's manual for **PIC**; a discussion of design issues and user experience may be found in [8]. The next section shows how to use **PIC** in the most simple way. Subsequent sections describe how to change the sizes of objects when the defaults are wrong, and how to change their positions when the standard positioning rules are wrong. An appendix

describes the language succinctly and more or less precisely.

2. Basics

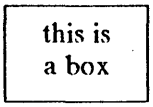
PIC provides boxes, lines, arrows, circles, ellipses, arcs, and splines (arbitrary smooth curves), plus facilities for positioning and labeling them. The picture below shows all of the fundamental objects (except for splines) in their default sizes:



Each picture begins with **.PS** and ends with **.PE**; between them are commands to describe the picture. Each command is typed on a line by itself. For example

```
.PS
box "this is" "a box"
.PE
```

creates a standard box (¾ inch wide, ½ inch high) and centers the two pieces of text in it:



Each line of text is a separate quoted string. Quotes are mandatory, even if the text contains no blanks. (Of course there needn't be any text at all.) Each line will be printed in the current size and font, centered horizontally, and separated vertically by the current **TROFF** line spacing.

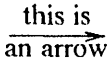
PIC does not center the drawing itself, but the default definitions of **.PS** and **.PE** in the **-ms** macro package do.

You can use **circle** or **ellipse** in place of **box**:

Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

```
.PS
arrow "this is" "an arrow"
.PE
```

produces



and

```
line "this is" "a line"
```

gives

this is
a line

Boxes and lines may be dashed or dotted; just add the word **dashed** or **dotted** after **box** or **line**.

Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clockwise by saying **arc cw**. So

```
line; arc; arc cw; arrow
```

produces

A spline might well do this job better; we will return to that shortly.

As you might guess,

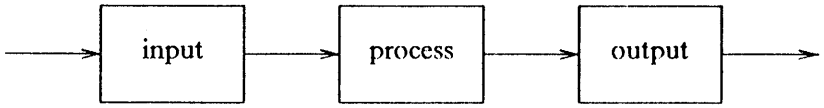
```
arc; arc; arc; arc
```

draws a circle, though not very efficiently.

Objects are normally drawn one after another, left to right, and connected at the obvious places. Thus the input

```
arrow; box "input"; arrow; box "process"; arrow; box "output"; arrow
```

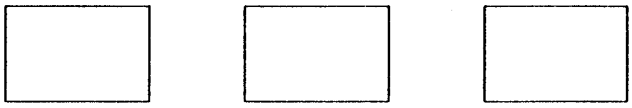
produces the figure



If you want to leave a space at some place, use **move**:

```
box; move; box; move; box
```

produces

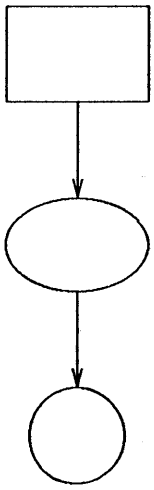


Notice that several commands can be put on a single line if they are separated by semicolons.

Although objects are normally connected left to right, this can be changed. If you specify a direction (as a separate object), subsequent objects will be joined in that direction. Thus

```
down; box; arrow; ellipse; arrow; circle
```

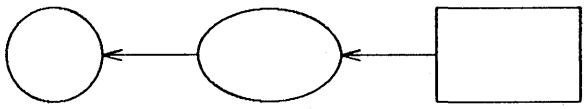
produces



and

```
left; box; arrow; ellipse; arrow; circle
```

produces



Each new picture begins going to the right.

Normally, figures are drawn at a fixed scale, with objects of a standard size. It is possible, however, to arrange that a figure be expanded to fit a particular width. If the **.PS** line contains a number, the drawing is forced to be that many inches wide, with the height scaled proportionately. Thus

```
.PS 3.5i
```

causes the picture to be 3.5 inches wide.

PIC is pretty dumb about the size of text in relation to the size of boxes, circles, and so on. There is as yet no way to say "make a box that just fits around this text" or "make this text fit inside this circle" or "draw a line as long as this text." All of these facilities are useful, so the limitations may go away in the fullness of time, but don't hold your breath. In the meantime, tight fitting of text can generally only be done by trial and error.

Speaking of errors, if you make a grammatical error in the way you describe a picture, **PIC** will complain and try to indicate where. For example, the invalid input

```
box arrow box
```

will draw the message

```
pic: syntax error near line 5, file -
 context is
      box arrow ^  box
```

The caret ^ marks the place where the error was first noted; it typically *follows* the word in error.

3. Controlling Sizes

This section deals with how to control the sizes of objects when the "default" sizes are not what is wanted. The next section deals with positioning them when the default positions are not right.

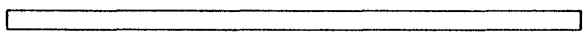
Each object that **PIC** knows about (boxes, circles, etc.) has associated dimensions, like height, width, radius, and so on. By default, **PIC** tries to choose sensible default values for these dimensions, so that simple pictures can be drawn with a minimum of fuss and bother. All of the figures and motions shown so far have been in their default sizes:

box	¾″ wide × ½″ high
circle	½″ diameter
ellipse	¾″ wide × ½″ high
arc	½″ radius
line or arrow	½″ long
move	½″ in the current direction

When necessary, you can make any object any size you want. For example, the input

```
box width 3i height 0.1i
```

draws a long, flat box



3 inches wide and 1/10 inch high. There must be no space between the number and the "**i**" that indicates a measurement in inches. In fact, the "**i**" is optional; all positions and dimensions are taken to be in inches.

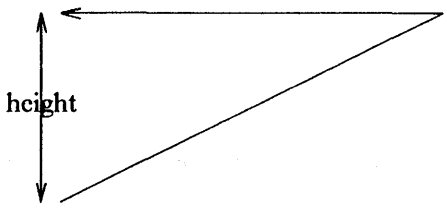
Giving an attribute like **width** changes only the one instance of the object. You can also change the default size for all objects of a particular type, as discussed later.

The attributes of **height** (which you can abbreviate to **ht**) and **width** (or **wid**) apply to boxes, circles, ellipses, and to the head on an arrow. The attributes of **radius** (or **rad**) and **diameter** (or **diam**) can be used for circles and arcs if they seem more natural.

Lines and arrows are most easily drawn by specifying the amount of motion from where one is right now, in terms of directions. Accordingly the words **up**, **down**, **left** and **right** and an optional distance can be attached to **line**, **arrow**, and **move**. For example,

```
.PS
line up 1i right 2i
arrow left 2i
move left 0.1i
line <-> down 1i "height"
.PE
```

draws

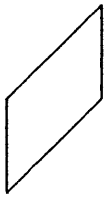


The notation <- > indicates a two-headed arrow; use - > for a head on the end and <- for one on the start. Lines and arrows are really the same thing; in fact, **arrow** is a synonym for **line - >**.

If you don't put any distance after **up**, **down**, etc., **PIC** uses the standard distance. So

```
line up right; line down; line down left; line up
```

draws the parallelogram



Warning: a very common error (which hints at a language defect) is to say

```
line 3i
```

A direction is needed:

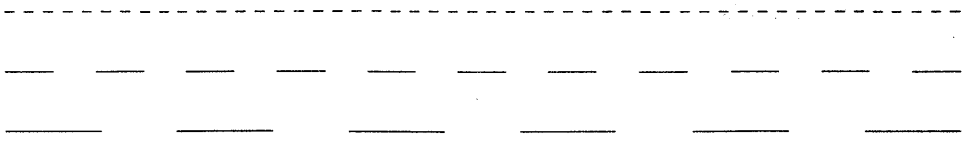
```
line right 3i
```

Boxes and lines may be dotted or dashed:

comes from

```
box dotted; line dotted; move; line dashed
```

If there is a number after **dot**, the dots will be that far apart. You can also control the size of the dashes (at least somewhat): if there is a length after the word **dashed**, the dashes will be that long, and the intervening spaces will be as close as possible to that size. So, for instance,



comes from the inputs (as separate pictures)

```
line right 5i dashed
line right 5i dashed 0.25i
line right 5i dashed 0.5i
line right 5i dashed 1i
```

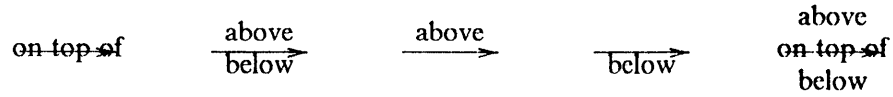
Sorry, but circles and arcs can't be dotted or dashed yet, and probably never will be.

You can make any object invisible by adding the word **invis(ible)** after it. This is particularly useful for positioning things correctly near text, as we will see later.

Text may be positioned on lines and arrows:

```
.PS
arrow "on top of"; move
arrow "above" "below"; move
arrow "above" above; move
arrow "below" below; move
arrow "above" "on top of" "below"
.PE
```

produces



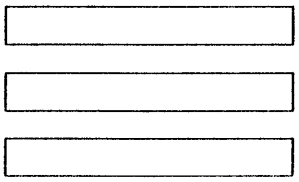
The "width" of an arrowhead is the distance across its tail; the "height" is the distance along the shaft. The arrowheads in this picture are default size.

As we said earlier, arcs go 90 degrees counterclockwise from where you are right now, and **arc cw** changes this to clockwise. The default radius is the same as for circles, but you can change it with the **rad** attribute. It is also easy to draw arcs between specific places; this will be described in the next section.

To put an arrowhead on an arc, use one of <-, -> or <->.

In all cases, unless an explicit dimension for some object is specified, you will get the default size. If you want an object to have the same size as the previous one of that kind, add the word **same**. Thus in the set of boxes given by

```
down; box ht 0.2i wid 1.5i; move down 0.15i; box same; move same; box same
```



the dimensions set by the first **box** are used several times; similarly, the amount of motion for the second **move** is the same as for the first one.

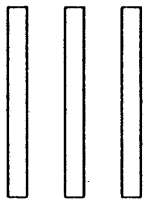
It is possible to change the default sizes of objects by assigning values to certain variables:

```
boxwid, boxht
linewid, lineht
dashwid
circlerad
arcrad
ellipsewid, ellipseht
movewid, moveht
arrowwid, arrowht     (These refer to the arrowhead.)
```

So if you want all your boxes to be long and skinny, and relatively close together,

```
boxwid = 0.1i; boxht = 1i
movewid = 0.2i
box; move; box; move; box
```

gives



PIC works internally in what it thinks are inches. Setting the variable **scale** to some value causes all dimensions to be scaled down by that value. Thus, for example, **scale = 2.54** causes dimensions to be interpreted as centimeters.

The number given as a width in the **.PS** line overrides the dimensions given in the picture; this can be used to force a picture to a particular size even when coordinates have been given in inches. Experience indicates that the easiest way to get a picture of the right size is to enter its dimensions in inches, then if necessary add a width to the **.PS** line.

4. Controlling Positions

You can place things anywhere you want; **PIC** provides a variety of ways to talk about places. **PIC** uses a standard Cartesian coordinate system, so any point or object has an x and y position. The first object is placed with its start at position 0,0 by default. The x,y position of a box, circle or ellipse is its geometrical center; the position of a line or motion is its beginning; the position of an arc is the center of the corresponding circle.

Position modifiers like **from**, **to**, **by** and **at** are followed by an x,y pair, and can be attached to boxes, circles, lines, motions, and so on, to specify or modify a position.

You can also use **up**, **down**, **right**, and **left** with **line** and **move**. Thus

```
.PS 2
box ht 0.2 wid 0.2 at 0,0 "1"
move to 0.5,0          # or "move right 0.5"
box "2" same           # use same dimensions as last box
move same              # use same motion as before
box "3" same
.PE
```

draws three boxes, like this:

Note the use of **same** to repeat the previous dimensions instead of reverting to the default values.

Comments can be used in pictures; they begin with a # and end at the end of the line.

Attributes like **ht** and **wid** and positions like **at** can be written out in any order. So

```
box ht 0.2 wid 0.2 at 0,0
box at 0,0 wid 0.2 ht 0.2
box ht 0.2 at 0,0 wid 0.2
```

are all equivalent, though the last is harder to read and thus less desirable.

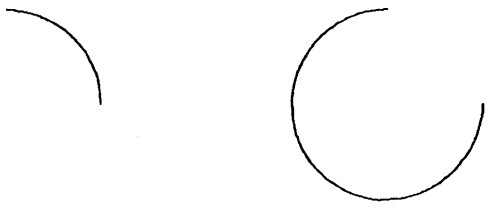
The **from** and **to** attributes are particularly useful with arcs, to specify the endpoints. By default, arcs are drawn counterclockwise,

```
arc from 0.5i,0 to 0,0.5i
```

is the short arc and

```
arc from 0,0.5i to 0.5i,0
```

is the long one:



If the **from** attribute is omitted, the arc starts where you are now and goes to the point given by **to**. The radius can be made large to provide flat arcs:

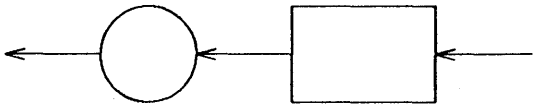
```
arc -> cw from 0,0 to 2i,0 rad 15i
```

produces

We said earlier that objects are normally connected left to right. This is an over-simplification. The truth is that objects are connected together in the direction specified by the most recent **up**, **down**, **left** or **right** (either alone or as part of some object). Thus, in

```
arrow left; box; arrow; circle; arrow
```

the **left** implies connection towards the left:



This could also be written as

```
left; arrow; box; arrow; circle; arrow
```

Objects are joined in the order determined by the last **up**, **down**, etc., with the entry point of the second object attached to the exit point of the first. Entry and exit points for boxes, circles and ellipses are on opposite sides, and the start and end of lines, motions and arcs. It's not entirely clear that this automatic connection and direction selection is the right design, but it seems to simplify many examples.

If a set of commands is enclosed in braces {...}, the current position and direction of motion when the group is finished will be exactly where it was when entered. Nothing else is restored. There is also a more general way to group objects, using **[** and **]**, which is discussed in a later section.

5. Labels and Corners

Objects can be labelled or named so that you can talk about them later. For example,

```
.PS
Box1:
    box ...
    # ... other stuff ...
    move to Box1
.PE
```

Place names have to begin with an upper case letter (to distinguish them from variables, which begin with lower case letters). The name refers to the "center" of the object, which is the geometric center for most things. It's the beginning for lines and motions.

Other combinations also work:

```
line from Box1 to Box2
move to Box1 up 0.1 right 0.2
move to Box1 + 0.2,0.1    # same as previous
line to Box1 - 0.5,0
```

The reserved name **Here** may be used to record the current position of some object, for example as

```
Box1: Here
```

Labels are variables – they can be reset several times in a single picture, so a line of the form

```
Box1: Box1 + 1i,1i
```

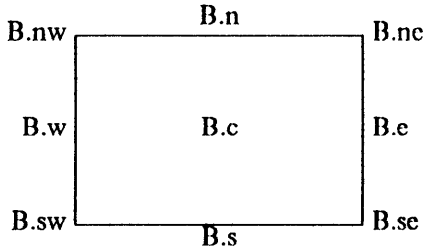
is perfectly legal.

You can also refer to previously drawn objects of each type, using the word **last**. For example, given the input

```
box "A"; circle "B"; box "C"
```

then '**last box**' refers to box **C**, '**last circle**' refers to circle **B**, and '**2nd last box**' refers to box **A**. Numbering of objects can also be done from the beginning, so boxes **A** and **C** are '**1st box**' and '**2nd box**' respectively.

To cut down the need for explicit coordinates, most objects have "corners" named by compass points:



The primary compass points may also be written as **.r**, **.b**, **.l**, and **.t**, for *right*, *bottom*, *left*, and *top*. The box above was produced with

```
.PS 1.5
B: box "B.c"
" B.e" at B.e ljust
" B.ne" at B.ne ljust
" B.se" at B.se ljust
"B.s" at B.s below
"B.n" at B.n above
"B.sw " at B.sw rjust
"B.w " at B.w rjust
"B.nw " at B.nw rjust
.PE
```

Note the use of **ljust**, **rjust**, **above**, and **below** to alter the default positioning of text, and of a blank with some strings to help space them away from a vertical line.

Lines and arrows have a **start**, an **end** and a center in addition to corners. (Arcs have only a center, a start, and an end.) There are a host of (i.e., too many) ways to talk about the corners of an object. Besides the compass points, almost any sensible combination of **left**, **right**, **top**, **bottom**, **upper** and **lower** will work. Furthermore, if you don't like the '.' notation, as in

```
last box.ne
```

you can instead say

```
upper right of last box
```

A longer statement like

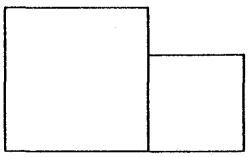
```
line from upper left of 2nd last box to bottom of 3rd last ellipse
```

begins to wear after a while, but it is descriptive. This part of the language is probably fat that will get trimmed.

It is sometimes easiest to position objects by positioning some part of one at some part of another, for example the northwest corner of one at the southeast corner of another. The **with** attribute in **PIC** permits this kind of positioning. For example,

```
box ht 0.75i wid 0.75i
box ht 0.5i wid 0.5i with .sw at last box.se
```

produces

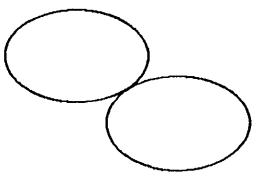


Notice that the corner after **with** is written **.sw**.

As another example, consider

```
ellipse; ellipse with .nw at last ellipse.se
```

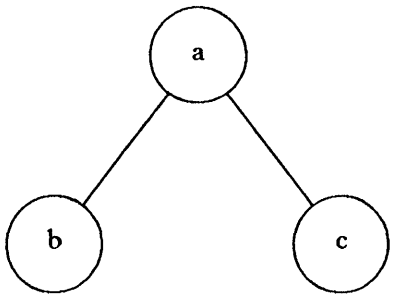
which makes



Sometimes it is desirable to have a line intersect a circle at a point which is not one of the eight compass points that **PIC** knows about. In such cases, the proper visual effect can be obtained by using the attribute **chop** to chop off part of the line:

```
circle "a"
circle "b" at 1st circle - (0.75i, 1i)
circle "c" at 1st circle + (0.75i, -1i)
line from 1st circle to 2nd circle chop
line from 1st circle to 3rd circle chop
```

produces



By default the line is chopped by **circlerad** at each end. This may be changed:

```
line ... chop r
```

chops both ends by *r*, and

```
line ... chop r1 chop r2
```

chops the beginning by *r1* and the end by *r2*.

There is one other form of positioning that is sometimes useful, to refer to a point some fraction of the way between two other points. This can be expressed in **PIC** as

```
fraction of the way between position1 and position2
```

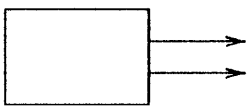
fraction is any expression, and *position1* and *position2* are any positions. You can abbreviate this rather windy phrase; "of the way" is optional, and the whole thing can be written instead as

```
fraction < position1 , position2 >
```

As an example,

```
box
arrow right from 1/3 of the way between last box.ne and last box.se
arrow right from 2/3 <last box.ne, last box.se>
```

produces



Naturally, the distance given by *fraction* can be greater than 1 or less than 0.

6. Variables and Expressions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. **PIC** variables let you parameterize your picture:

```
a = 0.5; b = 1

box wid a ht b
ellipse wid a/2 ht 1.5*b
move to Box1 - (a/2, b/2)
```

Expressions may use the standard operators +, −, *, /, and %, and parentheses for grouping.

Probably the most important variables are the predefined ones for controlling the default sizes of objects, listed in Section 4. These may be set at any time in any picture, and retain their values until reset.

You can use the height, width, radius, and x and y coordinates of any object or corner in an expression:

```
Box1.x              # the x coordinate of Box1
Box1.ne.y           # the y coordinate of the northeast corner of Box1
Box1.wid            # the width of Box1
Box1.ht             # and its height
2nd last circle.rad     # the radius of the 2nd last circle
```

Any pair of expressions enclosed in parentheses defines a position; furthermore such positions can be added or subtracted to yield new positions:

$$(x, y)$$
$$(x_1, y_1) + (x_2, y_2)$$

If p_1 and p_2 are positions, then

$$(p_1, p_2)$$

refers to the point

$$(p_1.x, p_2.y)$$

7. More on Text

Normally, text is centered at the geometric center of the object it is associated with. The attribute **ljust** causes the left end to be at the specified point (which means that the text lies to the right of the specified place!), and **rjust** puts the right end at the place. **above** and **below** center the text one half line space in the given direction.

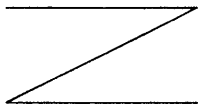
At the moment you can *not* compound text attributes: however natural it might seem, it is illegal to say "..." **above ljust**. This will be fixed eventually.

Text is most often an attribute of some other object, but you can also have self-standing text:

```
"this is some text" at 1,2 ljust
```

8. Lines and Splines

A "line" may actually be a path, that is, it may consist of connected segments like this:



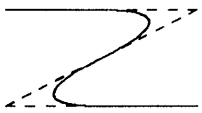
This line was produced by

```
line right 1i then down .5i left 1i then right 1i
```

A spline is a smooth curve guided by a set of straight lines just like the line above. It begins at the same place, ends at the same place, and in between is tangent to the mid-point of each guiding line. The syntax for a spline is identical to a (path) line except for using **spline** instead of **line**. Thus:

```
line dashed right 1i then down .5i left 1i then right 1i
spline from start of last line \
  right 1i then down .5i left 1i then right 1i
```

produces



(Long input lines can be split by ending each piece with a backslash.)

The elements of a path, whether for line or spline, are specified as a series of points, either in absolute terms or by **up**, **down**, etc. If necessary to disambiguate, the word **then** can be used to separate components, as in

```
spline right then up then left then up
```

which is not the same as

```
spline right up left up
```

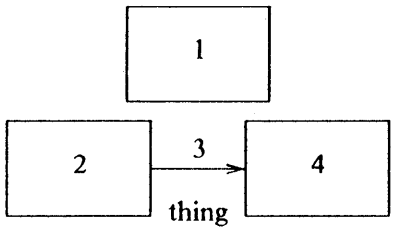
At the moment, arrowheads may only be put on the ends of a line or spline; splines may not be dotted or dashed.

9. Blocks

Any sequence of **PIC** statements may be enclosed in brackets **[...]** to form a block, which can then be treated as a single object, and manipulated rather like an ordinary box. For example, if we say

```
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"thing" at last [].s
```

we get



Notice that "last"-type constructs treat blocks as a unit and don't look inside for objects: "**last box.s**" refers to box 1, not box 2 or 4. You can use **last []**, etc., just like **last box**.

Blocks have the same compass corners as boxes (determined by the bounding box). It is also possible to position a block by placing either an absolute coordinate (like **0,0**) or an internal label (like **A**) at some external point, as in

```
[ ...; A: ...; ... ] with .A at ...
```

Blocks join with other things like boxes do (i.e., at the center of the appropriate side). It's not clear that this is the right thing to do, so it may change.

Names of variables and places within a block are local to that block, and thus do not affect variables and places of the same name outside. You can get at the internal place names with constructs like

```
last [].A
```

or

```
B.A
```

where **B** is a name attached to a block like so:

```
B : [ ... ; A: ...; ]
```

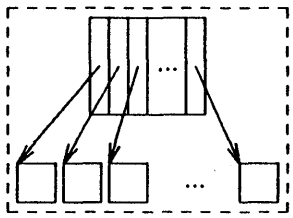
When combined with **define** statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is currently possible to look only one level deep with constructs like **B.A**, although **A** may be further qualified (i.e., **B.A.sw** or **top of B.A** are legal).

The following example illustrates most of the points made above about how blocks work:

```
h = .5i
dh = .02i
dw = .1i
[
	Ptr: [
		boxht = h; boxwid = dw
		A: box
		B: box
		C: box
		box wid 2*boxwid "..."
		D: box
	]
	Block: [
		boxht = 2*dw; boxwid = 2*dw
		movewid = 2*dh
		A: box; move
		B: box; move
		C: box; move
		box invis "..." wid 2*boxwid; move
		D: box
	] with .t at Ptr.s - (0,h/2)
	arrow from Ptr.A to Block.A.nw
	arrow from Ptr.B to Block.B.nw
	arrow from Ptr.C to Block.C.nw
	arrow from Ptr.D to Block.D.nw
]
box dashed ht last [].ht + dw wid last [].wid + dw at last []
```

This produces



10. Macros

PIC provides a rudimentary macro facility, the simple form of which is identical to that in **EQN**:

```
define name X replacement text X
```

defines *name* to be the *replacement text*; **X** is any character that does not appear in the replacement. Any subsequent occurrence of *name* will be replaced by *replacement text*.

Macros with arguments are also available. The replacement text of a macro definition may contain occurrences of **$1** through **$9**; these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings.

As an example, one might define a **square** by

```
define square X box ht $1 wid $1  $2 X
```

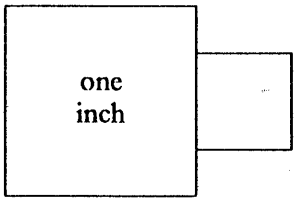
Then

```
square(1i, "one" "inch")
```

calls for a one inch square with the obvious label, and

```
square(0.5i)
```

calls for a square with no label:



Coordinates like *x,y* may be enclosed in parentheses, as in (*x,y*), so they can be included in a macro argument.

11. TROFF Interface

PIC is usually run as a **TROFF** preprocessor:

```
pic file | troff -ms
```

Run it before **EQN** and **TBL** if they are also present.

If the **.PS** line looks like

```
.PS < file
```

then the contents of **file** are inserted in place of the **.PS** line (whether or not the file contains **.PS** or **.PE**).

Other than this file inclusion facility, **PIC** copies the **.PS** and **.PE** lines from input to output intact, except that it adds two things right on the same line as the **.PS**:

```
.PS h w
```

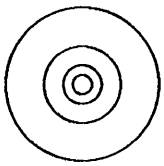
h and **w** are the picture height and width in units. The **-ms** macro package has simple definitions for **.PS** and **.PE** that cause pictures to be centered and offset a bit from surrounding text.

If ".**PF**" is used instead of **.PE**, the position after printing is restored to where it was before the picture started, instead of being at the bottom. "**F**" is for "flyback.")

Any input line that begins with a period is assumed to be a **TROFF** command that makes sense at that point; it is copied to the output at that point in the document. It is asking for trouble to add spaces or in any way fiddle with the line spacing here, but point size and font changes are generally harmless. So, for example,

```
.ps 24
circle radius .4i at 0,0
.ps 12
circle radius .2i at 0,0
.ps 8
circle radius .1i at 0,0
.ps 6
circle radius .05i at 0,0
.ps 10      \" don't forget to restore size
```

gives



It is also safe to muck about with sizes and fonts and local motions within quoted strings ("...") in **PIC**, so long as whatever changes are made are unmade before exiting the string. For example, to print text in Old English in size 8, use

```
ellipse "\s8\f(OESmile!\fP\s0"
```

This produces

This is essentially the same rule as applies in **EQN**.

There is a subtle problem with complicated equations inside **PIC** pictures – they come out wrong if **EQN** has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each equation the extra information **space 0**:

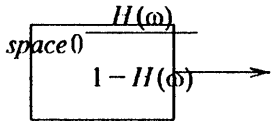
```
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
```

This produces

$-.$ *257text: S15* $<-$ $-$; *b=0,h=60,lf=1,rf=1*

$-.$ *257text: S15* $<-$ $-$; *b=0,h=60,lf=1,rf=1*

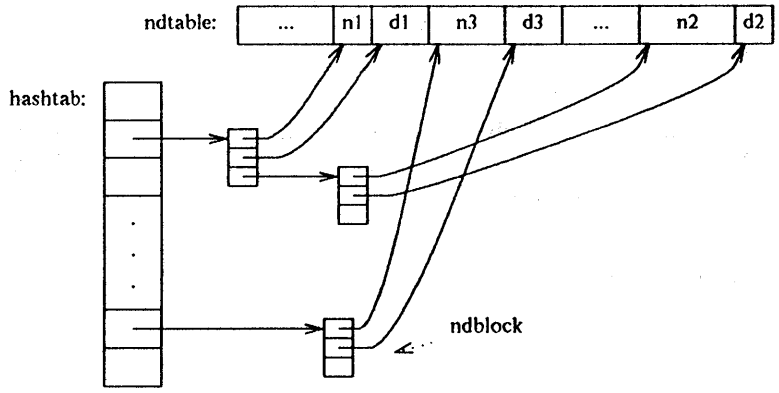
$-.$ *257text: S15* $<-$ $-$; *b=0,h=60,lf=1,rf=1*



PIC normally generates commands for a new version of **TROFF** that has operators for drawing graphical objects like lines, circles, and so on. As distributed, **PIC** assumes that its output is going to the Mergenthaler Linotron 202 unless told otherwise with the **-T** option. At present, the other alternatives are **-Tcat** (the Graphic Systems CAT, which does slanted lines and curves badly) and **-Taps** (the Autologic APS-5). It is likely that the option will already have been set to the proper default for your system, unless you have a choice of typesetters.

12. Some Examples

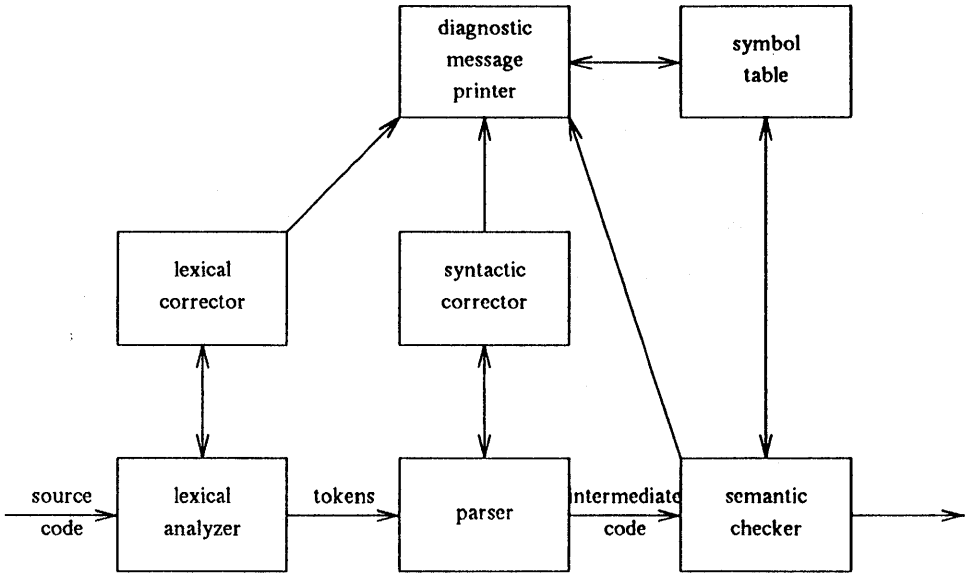
Herewith a handful of larger examples:



The input for the picture above was:

```
define ndblock X
        box wid boxwid/2 ht boxht/2
        down;  box same with .t at bottom of last box;   box same
X
boxht = .2i; boxwid = .3i; circlerad = .3i
down; box; box; box; box ht 3*boxht "." "." "."
L: box; box; box invis wid 2*boxwid "hashtab:" with .e at 1st box .w
right
Start: box wid .5i with .sw at 1st box.ne + (.4i,.2i) "..."
N1: box wid .2i "n1"; D1: box wid .3i "d1"
N3: box wid .4i "n3"; D3: box wid .3i "d3"
box wid .4i "..."
N2: box wid .5i "n2"; D2: box wid .2i "d2"
arrow right from 2nd box
ndblock
spline -> right .2i from 3rd last box then to N1.sw + (0.05i,0)
spline -> right .3i from 2nd last box then to D1.sw + (0.05i,0)
arrow right from last box
ndblock
spline -> right .2i from 3rd last box to N2.sw-(0.05i,.2i) to N2.sw + (0.05i,0)
spline -> right .3i from 2nd last box to D2.sw-(0.05i,.2i) to D2.sw + (0.05i,0)
arrow right 2*linewid from L
ndblock
spline -> right .2i from 3rd last box to N3.sw + (0.05i,0)
spline -> right .3i from 2nd last box to D3.sw + (0.05i,0)
circle invis "ndblock" at last box.e + (.7i,.2i)
arrow dotted from last circle to last box chop
box invis wid 2*boxwid "ndtable:" with .e at Start.w
```

This is the second example:



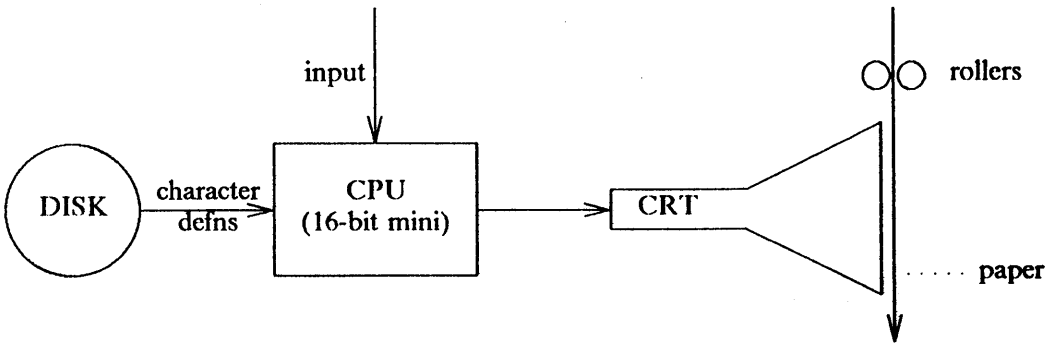
This is the input for the picture:

```
.PS 5
.ps 8
      arrow "source" "code"
LA:   box "lexical" "analyzer"
      arrow "tokens" above
P:    box "parser"
      arrow "intermediate" "code"
Sem:  box "semantic" "checker"
      arrow

      arrow <-> up from top of LA
LC:   box "lexical" "corrector"
      arrow <-> up from top of P
Syn:  box "syntactic" "corrector"
      arrow up
DMP:      box "diagnostic" "message" "printer"
      arrow <-> right  from right of DMP
ST:   box "symbol" "table"
      arrow from LC.ne to DMP.sw
      arrow from Sem.nw to DMP.se
      arrow <-> from Sem.top to ST.bot
.PE
```

There are eighteen objects (boxes and arrows) in the picture, and one line of **PIC** input for each; this seems like an acceptable level of verbosity.

The next example is the following:



Basic Digital Typesetter

This is the input for example 3:

```
.KS
.PS 5i
circle "DISK"
arrow "character" "defns"
box "CPU" "(16-bit mini)"
{ arrow <- from top of last box up "input " rjust }
arrow
CRT: "  CRT" ljust
line from CRT - 0,0.075 up 0.15 \
then right 0.5 \
then right 0.5 up 0.25 \
then down 0.5+0.15 \
then left 0.5 up 0.25 \
then left 0.5

Paper: CRT + 1.0+0.05,0
arrow from Paper + 0,0.75 to Paper - 0,0.5
{ move to start of last arrow down 0.25
  { move left 0.015; circle rad 0.05 }
  { move right 0.015; circle rad 0.05; "  rollers" ljust }
}
" paper" ljust at end of last arrow right 0.25 up 0.25
line left 0.2 dotted
.PE
.ce
Basic Digital Typesetter
.sp
.KE
```

13. Final Observations

PIC is not a sophisticated tool. The fundamental approach – Cartesian coordinates and real measurements – is not the easiest thing in the world to work with, although it does have the merit of being in some sense sufficient. Much of the syntactic sugar (or corn syrup) – corners, joining things implicitly, etc. – is aimed at making positioning and sizing automatic, or at least relative to previous things, rather than explicit.

Nonetheless, **PIC** does seem to offer some positive values. Most notably, it is integrated with the rest of the standard Unix document preparation software. In particular, it positions text correctly in relation to graphical objects; this is not true of any of the interactive graphical editors that I am aware of. It can even deal with equations in a natural manner, modulo the **space 0** nonsense alluded to above.

A standard question is, "Wouldn't it be better if it were interactive?" The answer seems to be both yes and no. If one has a decent input device (which I do not), interaction is certainly better for sketching out a figure. But if one has only standard terminals (at home, for instance), then a linear representation of a figure is better. Furthermore, it is possible to generate **PIC** input from a program: I have used **AWK**[9] to extract numbers from a report and generate the **PIC** commands to make histograms. This is hard to imagine with most of the interactive systems I know of.

In any case, the issue is far from settled; comments and suggestions are welcome.

Acknowledgements

I am indebted to Chris Van Wyk for ideas from several versions of **IDEAL**. He and Doug McIlroy have also contributed algorithms for line and circle drawing, and made useful suggestions on the design of **PIC**. Theo Pavlidis contributed the basic spline algorithm. Charles Wetherell pointed out reference [2] to me, and made several valuable criticisms on an early draft of the language and manual. The exposition in this version has been greatly improved by suggestions from Jim Blinn. I am grateful to my early users – Brenda Baker, Dottie Luciani, and Paul Tukey – for their suggestions and cheerful use of an often shaky and clumsy system.

References

1. J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 22
2. Brian W. Kernighan and Lorinda L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM*, vol. 18, no. 3, pp. 151-157, 1975.
3. DNL, M. E. Lesk, "Tbl – A Program to Format Tables," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 10
4. DNL, M. E. Lesk, "Some Applications of Inverted Indexes on the UNIX System," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 11
5. Christopher J. Van Wyk and C. J. Van Wyk, "A Graphics Typesetting Language," *SIGPLAN Symposium on Text Manipulation*, Portland, Oregon, June, 1981.
6. John C. Beatty, "PICTURE – A picture-drawing language for the Trix/Red Report Editor," Lawrence Livermore Laboratory Report UCID-30156, April 1977.
7. Anon., "V – A viewgraph generating language," Bell Laboratories internal memorandum, May 1979.
8. B. W. Kernighan, "PIC – A Language for Typesetting Graphics," *Software Practice & Experience*, vol. 12, no. 1, pp. 1-21, January, 1982.
9. A. V. Aho, P. J. Weinberger, and B. W. Kernighan, "AWK - A Pattern Scanning and Processing Language," *Software Practice and Experience*, vol. 9, pp. 267-280, April 1979.

Appendix A: PIC Reference Manual

Pictures

The top-level object in **PIC** is the "picture":

picture:
.PS *optional-width*
element-list
.PE

If *optional-width* is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion.

If instead the line is

.PS < f

the file **f** is inserted in place of the **.PS** line.

If **.PF** is used instead of **.PE**, the position after printing is restored to what it was upon entry.

Elements

An *element-list* is a list of elements (what else?); the elements are

element:
primitive attribute-list
placename : *element*
placename : *position*
variable = *expression*
direction
troff-command
{ *element-list* }
[*element-list*]

Elements in a list must be separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a # and terminated by a newline.

Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

The current position and direction of motion are saved upon entry to a {...} block and restored upon exit.

Elements within a block enclosed in **[...]** are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block are local to that block.

troff-command is any line that begins with a period. Such lines are assumed to make sense in the context where they appear; accordingly, if it doesn't work, don't call.

Primitives

The primitive objects are

primitive:
box
circle
ellipse
arc
line
arrow
move
spline
"any text at all"

arrow is a synonym for **line -> .**

Attributes

An *attribute-list* is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value. In the following, *e* is an expression and *opt-e* an optional expression.

attribute:

h(eigh)t *e*	wid(th) *e*
rad(ius) *e*	diam(eter) *e*
up *opt-e*	down *opt-e*
right *opt-e*	left *opt-e*
from *position*	to *position*
at *position*	with *corner*
by *e, e*	then
dotted *opt-e*	dashed *opt-e*
chop *opt-e*	-> <- <->
same	invis
text-list	

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. These are the currently meaningful attributes:

box:
    height, width, at, dotted, dashed, invis, same, *text*
circle and ellipse:
    radius, diameter, height, width, at, invis, same, *text*
arc:
    up, down, left, right, height, width, from, to, at, radius,
    invis, same, cw, <-, ->, <->, *text*
line, arrow
    up, down, left, right, height, width, from, to, by, then,
    dotted, dashed, invis, same, <-, ->, <->, *text*
spline:
    up, down, left, right, height, width, from, to, by, then,
    invis, <-, ->, <->, *text*
move:
    up, down, left, right, to, by, same, *text*
"text...":
    at, *text*

The attribute **at** implies placing the geometrical center at the specified place. For lines, splines and arcs, **height** and **width** refer to arrowhead size.

Text

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A *text-list* is a list of text items; a text item is a quoted string optionally followed by a positioning request:

text-item:
    "..."
    "..." center
    "..." ljust
    "..." rjust
    "..." above
    "..." below

If there are multiple text items for some primitive, they are centered vertically except as qualified. Positioning requests apply to each item independently.

Text items can contain **TROFF** commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

Positions and places

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:
    e, e
    place ± *e, e*
    (*position, position*)
    e [of the way] between *position* and *position*
    e < *position , position* >

The pair *e, e* may be enclosed in parentheses.

place:

placename *optional-corner*
corner placename
Here
corner of *nth primitive*
nth primitive optional-corner

A *corner* is one of the eight compass points or the center or the start or end of a primitive. (Not text!)

corner:

.n .e .w .s .ne .se .nw .sw
.t .b .r .l
.c .start .end

Each object in a picture has an ordinal number; *nth* refers to this.

nth:

*n*th
*n*th last

Since barbarisms like **1th** are barbaric, synonyms like **1st** and **3st** are accepted as well.

Variables

The built-in variables and their default values are:

boxwid 0.75i	boxht 0.5i
circlerad 0.25i	
ellipsewid 0.75i	ellipseht 0.5i
arcrad 0.25i	
linewid 0.5i	lineht 0.5i
movewid 0.5i	movewid 0.5i
arrowht 0.1i	arrowwid 0.05i
dashwid 0.1i	
scale 1	

These may be changed at any time, and the new values remain in force until changed again. Dimensions are divided by **scale** during output.

Expressions

Expressions in **PIC** are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expression:

e + *e*

e -- *e*

e * *e*

e / *e*

e % *e* (modulus)

- *e*

(*e*)

variable

number

place .x

place .y

place .ht

place .wid

place .rad

Definitions

The **define** statement is not part of the grammar.

define:

define *name* X *replacement text* X

Occurrences of **$1** through **$9** in the replacement text will be replaced by the corresponding arguments if **name** is invoked as

name(*arg1*, *arg2*, ...)

Non-existent arguments are replaced by null strings. *Replacement text* may contain newlines.

This page intentionally left blank.

December 1987

APPENDICES

The following appendices are provided:

- **Appendix A. Software Description**

  lists those few functions of 4.3BSD that are *not* supported in this distribution of IBM/4.3.

- **Appendix B. Graphics Manual Pages**

  contains manual pages for the graphics routines used by the C subroutine interface described in Volume II.

- **Appendix C. High C Programmer's Guide**

  contains a guide for programming in C, using the High C compiler from MetaWare Incorporated.

This page intentionally left blank.

Appendix A. Software Description

This appendix contains a listing of unsupported functions found in IBM/4.3.

1. UNSUPPORTED FUNCTIONS

The following sections list the functions of 4.3BSD for the VAX that are not supported by IBM/4.3.

1.1. Section 1: Commands and Application Programs.

Man Page	Name	Section and Description
fp	fp	(1) functional programming language compiler/interpreter
gcore	gcore	(1) get core images of running processes
lisp	lisp	(1) Lisp interpreter
liszt	liszt	(1) compile a Franz Lisp programx
lxref	lxref	(1) Lisp cross reference program
pc	pc	(1) Pascal compiler
pdx	pdx	(1) Pascal debugger
pi	pi	(1) Pascal interpreter code translator
pix	pix	(1) Pascal interpreter and executor
pmerge	pmerge	(1) Pascal file merger
px	px	(1) Pascal interpreter
pxp	pxp	(1) Pascal execution profiler
pxref	pxref	(1) Pascal cross-reference program
sccs	sccs	(1) front end for the sccs subsystem
tc	tc	(1) phototypesetter simulator
tcopy	tcopy	(1) copy a mag tape
tk	tk	(1) paginator for the Tektronix 4014
tp	tp	(1) manipulate tape archive
vlp	vlp	(1) format Lisp programs to be printed with nroff, vtroff, or troff
vwidth	vwidth	(1) make troff width table for a font

1.2. Section 2: System Calls

NONE

1.3. Section 3: C Library Subroutines

NONE

1.4. Section 3F: FORTRAN Library

Man Page	Name	Section and Description
plot	arc	(3F) f77 library interface to *plot*(3X) libraries
plot	box	(3F) f77 library interface to *plot*(3X) libraries
plot	circle	(3F) f77 library interface to *plot*(3X) libraries
plot	clospl	(3F) f77 library interface to *plot*(3X) libraries
plot	cont	(3F) f77 library interface to *plot*(3X) libraries
plot	erase	(3F) f77 library interface to *plot*(3X) libraries
plot	label	(3F) f77 library interface to *plot*(3X) libraries
plot	line	(3F) f77 library interface to *plot*(3X) libraries

plot	linemd	(3F) f77 library interface to *plot*(3X) libraries
plot	move	(3F) f77 library interface to *plot*(3X) libraries
plot	openpl	(3F) f77 library interface to *plot*(3X) libraries
plot	plot	(3F) f77 library interface to *plot*(3X) libraries
plot	point	(3F) f77 library interface to *plot*(3X) libraries
plot	space	(3F) f77 library interface to *plot*(3X) libraries
random	drandm	(3F) better random number generator
random	irandm	(3F) better random number generator
random	random	(3F) better random number generator

1.5. Section 3G: AED Graphics Subroutines

NONE

1.6. Section 3M: Math Library

Man Page	Name	Section and Description
infnan	infnan	(3M) signals invalid floating point operations on a VAX (temporary)

1.7. Section 3N: Internet Network Library

Man Page	Name	Section and Description
ns	ns_addr	(3N) Xerox NS(tm) address conversion routines
ns	ns_ntoa	(3N) Xerox NS(tm) address conversion routines

1.8. Section 3S: C Standard I/O Library Subroutines

NONE

1.9. Section 3X: Other Libraries

Man Page	Name	Section and Description
lib2648	lib2648	(3X) subroutines for the HP 2648 graphics terminal
plot	arc	(3X) graphics interface
plot	circle	(3X) graphics interface
plot	closepl	(3X) graphics interface
plot	cont	(3X) graphics interface
plot	erase	(3X) graphics interface
plot	label	(3X) graphics interface
plot	line	(3X) graphics interface
plot	linemod	(3X) graphics interface
plot	move	(3X) graphics interface
plot	openpl	(3X) graphics interface
plot	plot	(3X) graphics interface
plot	point	(3X) graphics interface
plot	space	(3X) graphics interface

1.10. Section 3C: Compatibility Library Subroutines

NONE

1.11. Section 4: Special Files

Man Page	Name	Section and Description
acc	acc	(4) ACC LH/DH IMP interface
ad	ad	(4) Data Translation A/D converter
css	css	(4) DEC IMP-11A LH/DH IMP interface
crl	crl	(4) VAX 8600 console RL02 interface
ct	ct	(4) phototypesetter interface
ddn	ddn	(4) DDN Standard Index.25 IMP Interface
de	de	(4) DEC DEUNA 10 Mb/s Ethernet Interface
dh	dh	(4) DH-11/DM-11 communications multiplexer
dhu	dhu	(4) DHU-11 communications multiplexer
dmc	dmc	(4) DEC DMC-11/DMR-11 point-to-point communications device
dmf	dmf	(4) DMF-32, terminal multiplexer
dmz	dmz	(4) DMZ-32 terminal multiplexer
dn	dn	(4) DN-11 autocall unit interface
dz	dz	(4) DZ-11 communications multiplexer
ec	ec	(4) 3Com 10 Mb/s Ethernet interface
en	en	(4) Xerox 3 Mb/s Ethernet interface
ex	ex	(4) Excelan 10 Mb/s Ethernet interface
fl	fl	(4) console diskette interface
hdh	hdh	(4) ACC IF-11/HDH IMP interface
hk	hk	(4) RK6-11/RK06 and RK07 moving head disk
hp	hp	(4) MASSBUS disk interface
ht	ht	(4) TM-03/TE-16,TU-45,TU-77 MASSBUS magtape interface
hy	hy	(4) Network Systems Hyperchannel interface
idp	idp	(4P) Xerox Internet Datagram Protocol
ik	ik	(4) Ikonas frame buffer, graphics device interface
il	il	(4) Interlan 10 Mb/s Ethernet interface
imp	imp	(4) 1822 network interface
ix	ix	(4) Interlan NP100 10 Mb/s Ethernet interface
kg	kg	(4) KL-11/DL-11W line clock
mt	mt	(4) TM78/TU-78 MASSBUS magtape interface
np	np	(4) Interlan NP100 Mb/s Ethernet interface
ns	ns	(4F) Xerox Network Systems(tm) protocol family
nsip	nsip	(4) software network interface encapsulating ns packets in ip packets
pcl	pcl	(4) DEC CSS PCL-11 B Network Interface
ps	ps	(4) Evans and Sutherland Picture System 2 graphics device interface
qe	qe	(4) DEC DEQNA Q-bus 10 Mb/s Ethernet interface
rx	rx	(4) DEC RX02 diskette interface
spp	spp	(4P) Xerox Sequenced Packet Protocol
tm	tm	(4) TM-11/TE-10 magtape interface
tmscp	tmscp	(4) DEC TMSCP magtape interface
ts	ts	(4) TS-11 magtape interface
tu	tu	(4) VAX-11/730 and VAX-11/750 TU58 console cassette interface
uda	uda	(4) UDA-50 disk controller interface
up	up	(4) unibus storage module controller/drives
ut	ut	(4) UNIBUS TU45 tri-density tape drive interface

uu	uu	(4) TU58/DECtape II UNIBUS cassette interface
va	va	(4) Benson-Varian interface
vp	vp	(4) Versatec interface
vv	vv	(4) Proteon proNET 10 Megabit ring

1.12. Section 5: File Formats

Man Page	Name	Section and Description
tp	tp	(5) DEC/mag tape formats
vfont	vfont	(5) font formats for the Benson-Varian or Versatec

1.13. Section 6: Games

Man Page	Name	Section and Description
aardvark	aardvark	(6) yet another exploration game
adventure	adventure	(6) an exploration game
arithmetic	arithmetic	(6) provide drill in number facts
backgammon	backgammon	(6) the game of backgammon
banner	banner	(6) print large banner on printer
battlestar	battlestar	(6) a tropical adventure game
bcd	bcd	(6) convert to antique media
boggle	boggle	(6) the game of boggle
canfield	canfield	(6) the solitaire card game Canfield
canfield	cfscores	(6) the solitaire card game Canfield
chess	chess	(6) the game of chess
ching	ching	(6) the book of changes and other cookies
cribbage	cribbage	(6) the card game cribbage
doctor	doctor	(6) interact with a psychoanalyst
fish	fish	(6) play Go Fish
fortune	fortune	(6) print a random, hopefully interesting, adage
hangman	hangman	(6) computer version of the game hangman
hunt	hunt	(6) a multiplayer multiterminal game
mille	mille	(6) play Mille Bournes
monop	monop	(6) the game of Monopoly
number	number	(6) convert Arabic numerals to English
quiz	quiz	(6) test your knowledge
rain	rain	(6) animated raindrops display
rogue	rogue	(6) exploring the dungeons of doom
sail	sail	(6) mulituser wooden ships and iron men
snake	snake	(6) display chase game
snake	snscore	(6) display chase game
trek	trek	(6) trekkie game
worm	worm	(6) the growing worm game
worms	worms	(6) animate worms on a display terminal
wump	wump	(6) the game of hunt-the-wumpus
zork	zork	(6) the game of dungeon

1.14. Section 7: Miscellaneous

NONE

1.15. Section 8: System Maintenance

Man Page	Name	Section and Description
analyze	analyze	(8) virtual UNIX postmortem crash analyzer
arff	arff	(8R) archiver and copier for diskette
arff	flcopy	(8) archiver and copier for diskette

bad144	bad144	(8) read/write DEC standard 144 bad sector information
drtest	drtest	(8) standalone disk test program
implog	implog	(8C) IMP log interpreter
implogd	implogd	(8C) IMP logger process
rxformat	rxformat	(8V) format diskettes
XNSrouted	XNSrouted	(8C) NS Routing Information Protocol daemon

Appendix B. Graphics Manual Pages for the IBM Academic Information Systems Experimental Display

This section contains the manual pages for section 3G; they describe the display graphics subroutines. You may want to file these manual pages in Volume I.

- intro (3G)
- circle (3G)
- clip (3G)
- color (3G)
- copy (3G)
- cursor (3G)
- dash (3G)
- font (3G)
- force (3G)
- image (3G)
- init (3G)
- line (3G)
- log (3G)
- merge (3G)
- move (3G)
- query (3G)
- read (3G)
- run (3G)
- string (3G)
- tile (3G)
- width (3G)

This page intentionally left blank.

NAME

intro – introduction to display graphics subroutines

DESCRIPTION

This section describes the subroutines that are part of the interface for the IBM Academic Information Systems experimental display (herein after called "the experimental display"). The subroutines are graphics routines for controlling the experimental display in all-points addressable mode.

The interface described in this section provides access to a set of functions designed to support a window manager, and is composed primarily of subroutines, as distinguished from functions. A typical subroutine uses parameters to receive input and return output. C passes parameters by value; to call a subroutine which returns information, you must supply an address for the returning value as the parameter.

Calls that supply an address for return in this package should usually supply the address of a *short* (16-bit) integer. Calls that pass integer values can usually get by with either *short* or *int*. See the individual routines.

Many of the subroutines do return a value as a function would, generally for error return codes and special case handling. It is strongly recommended that applications monitor return codes to prevent bizarre events and possibly more severe errors.

When linking, specify *-laed* to pick up the experimental-display library.

All subroutines use screen coordinates with the origin in the upper left corner of the experimental display.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
VI_ALine	line.3g	draw a line to an absolute location
VI_AMove	move.3g	move the current point to an absolute location
VI_Circle	circle.3g	draw a circle
VI_Clip	clip.3g	set clipping window
VI_Color	color.3g	change screen color
VI_Copy	copy.3g	copy an area
VI_Dash	dash.3g	set line dash pattern
VI_DisCur	cursor.3g	disable cursor
VI_DropFont	font.3g	release font
VI_EnCur	cursor.3g	enable cursor
VI_FDefnCur	cursor.3g	set cursor pattern from file
VI_FImage	image.3g	draw an image from a file
VI_Font	font.3g	select font
VI_Force	force.3g	force output of graphics orders
VI_Fread	read.3g	read experimental-display data into a file
VI_GetFont	font.3g	load a font into memory
VI_Init	init.3g	initialize the subroutine interface
VI_Login	log.3g	begin logging subroutine calls
VI_Logout	log.3g	close a log file
VI_MDefnCur	cursor.3g	set cursor pattern from memory
VI_Merge	merge.3	set merge mode
VI_MImage	image.3g	draw an image from memory
VI_MRead	read.3g	read experimental-display data into memory
VI_PosnCur	cursor.3g	set cursor position
VI_QClip	query.3	query clipping rectangle
VI_QColor	query.3g	query current color
VI_QDash	query.3g	query dash pattern

VI_QFont	query.3g	query font
VI_QMerge	query.3g	query merge mode
VI_QPoint	query.3g	query current point
VI_QWidth	query.3g	query line width
VI_RLine	line.3g	draw a line to a relative location
VI_RMove	move.3g	move the current point to a relative location
VI_Run	run.3g	process a log file
VI_String	string.3g	draw a string
VI_Term	init.3g	terminate the subroutine interface
VI_Tile	tile.3g	tile a rectangle
VI_Width	width.3g	set line width

FILES

/usr/lib/aed/whim.aed
/usr/lib/aed/pcfont.fnt
/usr/lib/libaed.a
/usr/src/usr.lib/libaed/examples
/dev/aed

NOTE

These subroutines apply only to the IBM Academic Information Systems experimental display.

SEE ALSO

"The C Subroutine Interface for the IBM Academic Information Systems Experimental Display" in Volume II, Supplementary Documents.

NAME

VI_Circle – draw a circle

SYNOPSIS

```
VI_Circle(radius)
    int radius;          /* circle radius */
```

DESCRIPTION

VI_Circle draws a circle with the specified radius and the current point as its center. The current point is unchanged.

NOTE

VI_Circle applies only to the IBM Academic Information Systems experimental display. The line attributes *VI_Dash* and *VI_Width* do not apply to *VI_Circle*.

Nothing is drawn if the radius is less than or equal to zero. You cannot use concentric circles to do a solid area fill.

This page intentionally left blank.

NAME

VI_Clip – set clipping window

SYNOPSIS

```
VI_Clip(lx,ly,hx,hy)
        int lx,ly;      /* top left corner of clipping area */
        int hx,hy;      /* bottom right corner of clipping area */
```

DESCRIPTION

VI_Clip specifies that subsequent primitives drawn on the screen are to be clipped to the specified area. It is the user's responsibility to ensure the sensibility of the window definition. The clipping window is initially set to the whole screen.

NOTE

VI_Clip applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

query(3G)

This page intentionally left blank.

NAME

VI_Color – change screen color

SYNOPSIS

```
VI_Color(color)
        int color;              /* new color, true for white */
```

DESCRIPTION

VI_Color sets the color of the screen to the specified value: 0 means that bits having the binary value "0" will be black on the screen; 1 means that bits having the binary value "1" will be black on the screen. If this value is different from the previous value, the screen will be inverted, so as to make the change transparent to the application. The screen color is initially white 1's on black 0's, color 0.

NOTE

VI_Color applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

query(3G)

This page intentionally left blank.

NAME

VI_Copy – copy an area

SYNOPSIS

```
VI_Copy(sx,sy,tx,ty,wd,ht,merge)
    int sx,sy;              /* source top-left */
    int tx,ty;              /* target top-left */
    int wd,ht;          /* rectangle dimensions */
    int merge;          /* merge mode */
```

DESCRIPTION

VI_Copy duplicates the rectangle at *sx,sy* with the dimensions *wd,ht* to the point *tx,ty*. The copied bits are merged with the target area using the specified merge mode, not the merge mode set by *merge*(3G). See *merge*(3G) for a description of merge modes.

Both the source and destination rectangles must be completely on the screen. The current setting of the clipping window is ignored.

NOTE

VI_Copy applies only to the IBM Academic Information Systems experimental display.

VI_Copy cannot copy an area onto itself with a mode change, e.g. for highlighting. A fast way to highlight is to use *VI_Merge* with XOR mode and *VI_Tile*.

SEE ALSO

merge (3G)

This page intentionally left blank.

NAME

VI_MDefnCur, VI_FDefnCur, VI_EnCur, VI_DisCur, VI_PosnCur – control the display cursor

SYNOPSIS

```
VI_MDefnCur(xoff,yoff,black,white)
        int xoff;               /* x offset of cursor center */
        int yoff;               /* y offset of cursor center */
        unsigned short *black;  /* first byte of black mask */
        unsigned short *white;  /* first byte of white mask */

VI_FDefnCur(filename)
        char *filename;         /* name of cursor definition file */

VI_EnCur()

VI_DisCur()

VI_PosnCur(x,y)
        int x,y;                        /* new cursor position */
```

DESCRIPTION

These subroutines allow programs to control the display cursor by defining it, enabling and disabling it, and changing its position. Disabling and reenabling the cursor do not affect its pattern or position. Because the display maintains the cursor separately from the display buffer, the cursor does not have to be removed when a graphics primitive intersects its position. Initially the cursor is transparent and disabled, and is positioned at the center of the screen.

VI_MDefnCur Sets the cursor as specified. *xoff,yoff* is the displacement of the cursor pattern from the current position of the cursor. For example, a value of (32,32) would center the cursor pattern around the current point. The cursor pattern itself is a 64-by-64 bit image, with two planes. A 1 in the black plane indicates that that bit of the cursor should be black. A 1 in the white plane indicates that the cursor should be white in that position. If a bit has a 0 in both planes, the cursor is transparent in that position. If a bit is 1 in both planes, the cursor is white. The two planes are images in the same format as accepted by *MImage,* and must be 64-by-64, or 512 bytes each.

VI_FDefnCur Sets the cursor to the definition in the specified file. The file has the format shown below; the fields are explained under *MDefnCur.*

Offset (bytes)	Description
0	XOFF
2	YOFF
4	BLACK bit pattern
516	WHITE bit pattern

See the description of *MDefnCur* for a description of the fields.

VI_EnCur Enables the cursor and displays it if it is not already present.

VI_DisCur Disables the cursor and removes it from the screen if it is present.

VI_PosnCur Moves the cursor to the specified position. It cannot be moved off the screen.

NOTE

VI_Cursor applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

image(3G)

This page intentionally left blank.

NAME

VI_Dash – set line dash pattern

SYNOPSIS

```
VI_Dash(dash,dashlen)
    unsigned short dash;        /* dash pattern */
    short dashlen;              /* dash pattern length */
```

DESCRIPTION

If no dash pattern has been set, lines drawn with the *VI_RLine* and *VI_ALine* subroutines described under *line*(3G) are solid lines of 1's. If a pattern has been set, the bits of the pattern word are used in sequence whenever the vector generator would normally output a 1. Setting a pattern of 0x5555 produces a very acceptable dotted line. Other patterns may be used to vary the size of dashes in the line. The length of the pattern can range from 1 to 16 bits. The pattern bits should be left-justified. Setting the pattern length to 0 specifies a return to solid lines. The line dash pattern is initially set to solid 1's.

SEE ALSO

line(3G), merge(3G), query(3G), width(3G)

NOTE

VI_Dash applies only to the IBM Academic Information Systems experimental display. *VI_Dash* does not support *VI_Circle*.

This page intentionally left blank.

NAME

VI_Font, VI_GetFont, VI_DropFont – select and manipulate fonts

SYNOPSIS

```
VI_Font(fontid)
        int fontid;             /* font ID */

VI_GetFont(name,fontid)
        char *name;             /* font name */
        short *fontid;          /* font ID */

VI_DropFont(fontid)
        int fontid;             /* ID of font to release */
```

DESCRIPTION

Fonts are stored in files, which are loaded into the workstation memory when requested by applications using *VI_GetFont*. Once a font is loaded, it is kept in memory until the program ends, unless explicitly dropped with *VI_DropFont*.

VI_GetFont — Loads the specified font into memory, if it is not already present. If the font is successfully loaded, the font ID is returned. Setting the current font to this ID with *VI_Font* causes subsequent strings to be displayed in the font.

VI_Font — Selects the font with the specified font ID. Font IDs range from 0 to 255 and are returned by calls to *VI_GetFont*.

VI_DropFont — Drops the specified font from memory. The application should not attempt to use the font ID again. If the font is required, a new font ID should be generated by a request to *VI_GetFont*.

NOTE

VI_Font applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

string(3G)

DIAGNOSTICS

If *VI_GetFont* returns a font ID of 0, either the font could not be found, or it did not fit in memory. If the font did not fit in memory, a message will be sent to *stderr*.

This page intentionally left blank.

NAME

VI_Force – force output of graphics orders

SYNOPSIS

VI_Force()

DESCRIPTION

Commands built with subroutines described in "Setting Graphics Parameters" and "Issuing Graphics Primitives" in "The C Subroutine Interface for the IBM Academic Information Systems Experimental Display" generally do not send their output to the screen immediately. Instead the output remains in a buffer until the buffer is full, when its output is sent to the screen. Use *VI_Force* to force output in the current buffer to be transmitted before the buffer is full.

NOTE

VI_Force applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

init(3G)

This page intentionally left blank.

This page intentionally left blank.

NAME

VI_Init, VI_Term – initialize and terminate the subroutine interface

SYNOPSIS

```
VI_Init(wd,ht)
        short *wd,*ht;          /* screen dimensions */

VI_Term()
```

DESCRIPTION

These functions initialize and terminate the subroutine interface.

VI_Init Initializes the display and returns the dimensions of the screen. The display currently has a width of 1024 bits and a height of 800 bits. *VI_Init* must be the first call. The top left point is (0,0); the bottom right point is (1023,799).

VI_Term Completes processing, closes any log files, and forces transmission of the graphics buffer to the display.

FILES

/dev/aed
/usr/lib/aed/whim.aed
/usr/lib/aed/pcfont.fnt

NOTE

VI_Init applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

force(3G), log(3G)

NAME

VI_MImage, VI_FImage – draw an image

SYNOPSIS

```
VI_MImage(wd,ht,data)
        int wd,ht;                      /* dimensions of image */
        unsigned short *data;           /* first byte of image */

VI_FImage(filename)
        char *filename;         /* file name of image to draw */
```

DESCRIPTION

These functions draw an image from memory or from a file. The current point is unchanged. The image data should be in scanline order, from top to bottom, with each scanline padded to the next 16-bit word. For example, for a width of WD and height of HT, there should be 2*HT(WD+15)/16 bytes of image data.

VI_MImage Draws an image of the specified dimensions whose top left corner is at the current point. *data* must be the first byte of an image large enough to fill the rectangle specified by *wd* and *ht,* or an addressing error may result.

VI_FImage Draws the image contained in the specified file, placing its top left corner at the current point. The image file must have the following format:

Offset (bytes)	Description
0	The width of the image
2	The height of the image
4	Image data

NOTE

VI_Image applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

read(3G)

This page intentionally left blank.

NAME

VI_Init, VI_Term − initialize and terminate the subroutine interface

SYNOPSIS

VI_Init(wd,ht)
short *wd,*ht; /* screen dimensions */

VI_Term()

DESCRIPTION

These functions initialize and terminate the subroutine interface.

VI_Init Initializes the display and returns the dimensions of the screen. The display currently has a width of 1024 bits and a height of 800 bits. *VI_Init* must be the first call. The top left point is (0,0); the bottom right point is (1023,799).

VI_Term Completes processing, closes any log files, and forces transmission of the graphics buffer to the display.

FILES

/dev/aed
/usr/lib/aed/whim.aed
/usr/lib/aed/pcfont.fnt

NOTE

VI_Init applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

force(3G), log(3G)

This page intentionally left blank.

NAME

VI_ALine, VI_RLine – draw a line

SYNOPSIS

```
VI_ALine(x,y)
        int x,y;        /* end point of line */
VI_RLine(dx,dy)
        int dx,dy;      /* displacement to end point */
```

DESCRIPTION

These functions draw a line to an absolute or a relative location. A line is normally of 1's, and is merged with the window data according to the current merge mode.

VI_ALine Draws a line from the current point to the specified point (the line's end point) according to the current values of the merge, width, and dash pattern parameters. The specified point becomes the current point.

VI_RLine Draws a line from the current point to the current point displaced by the specified values, according to the current values of the merge, width, and dash pattern parameters. The current point is incremented by the displacement.

NOTE

VI_Line applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

clip(3G), dash(3G), merge(3G), width(3G)

This page intentionally left blank.

NAME

VI_Login, VI_Logout – begin logging subroutine calls and close a log file

SYNOPSIS

```
int VI_Login(filename)
    char *filename;          /* file to log to */

int VI_Logout()
```

DESCRIPTION

These subroutines begin logging subroutine calls and close the log file.

VI_Login Specifies that subsequent subroutine calls are to be echoed into the specified file. If a log file is already open, *VI_Login* closes it before opening the new file; *VI_Login* overwrites an existing file. All orders to the display are logged until a logout call *(VI_Logout)* is issued. The log file may later be executed from within a program using *VI_Run* or on its own using *aedrunner*(1). It may also be examined with *aedjournal*(1).

VI_Logout Closes the log file.

NOTE

VI_Log applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

aedjournal(1), aedrunner(1), init(3G), run(3G)

"The C Subroutine Interface for the IBM Academic Information Systems Experimental Display" in Volume II.

DIAGNOSTICS

VI_Login returns a negative value if there is an error, and a nonnegative value if the call is successful.

VI_Logout returns one of three values:

Value	Meaning
0	Normal completion
-1	Error in closing file
-2	No file found to close

This page intentionally left blank.

NAME

VI_Merge – set merge mode

SYNOPSIS

```
VI_Merge(merge)
    int merge;          /* merge mode */
```

DESCRIPTION

The merge mode is a number from 0 to 15 that specifies how the bits generated by primitives are to be combined with bits already on the screen, as shown in the following table:

Merge Mode	Meaning
0	OFF
1	NOR
2	NOT DATA AND SCREEN
3	NOT DATA
4	DATA AND NOT SCREEN
5	NOT SCREEN
6	XOR (NEQ)
7	NAND
8	AND
9	EQ
10	SCREEN (ignore)
11	NOT DATA OR SCREEN
12	DATA (replace)
13	DATA OR NOT SCREEN
14	OR
15	ON

The merge mode is initially set to 12, for replace mode. Data bits replace screen bits. The merge mode is simply an encoding of the logical function used to combine screen bits and data bits. Encoding the desired result of each of the combinations in the table below generates the merge mode that should be used to get that effect. For example, to *or* the data you are adding with the data already present on the screen, you would use a merge mode of 14:

Data Bit	1	1	0	0	
Screen Bit	1	0	1	0	
Example:					
OR mode	1	1	1	0	= 14

NOTE

VI_Merge applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

circle(3G), color(3G), line(3G), query(3G)

This page intentionally left blank.

NAME

VI_AMove, VI_RMove − move the current point

SYNOPSIS

```
VI_AMove(x,y)
        int x,y;        /* new point */
VI_RMove(dx,dy)
        int dx,dy;      /* displacement from old point */
```

DESCRIPTION

These functions move the current point; they do not change the screen. The current point is initially set to (0,0).

VI_AMove Moves the current point to the specified coordinates.

VI_RMove Moves the current point by the specified displacement.

NOTE

VI_Move applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

query(3G)

This page intentionally left blank.

NAME

VI_QClip, VI_QColor, VI_QDash, VI_QFont, VI_QMerge, VI_QPoint, VI_QWidth – query graphics parameters

SYNOPSIS

```
VI_QClip(lx,ly,hx,hy)
        short *lx,*ly;          /* top left corner of clipping area */
        short *hx,*hy;          /* bottom right corner */

VI_QColor(color)
        short *color;           /* current color, true for white */

VI_QDash(dash,dashlen)
        unsigned short *dash;           /* dash pattern */
        short *dashlen;         /* length of dash pattern */

VI_QFont(fontid,fontname)
        short *fontid;          /* current font ID */
        char *fontname; /* current font name */

VI_QMerge(merge)
        short *merge;           /* current merge mode */

VI_QPoint(x,y)
        short *x,*y             /* current point */

VI_QWidth(width)
        short *width;           /* line width */
```

DESCRIPTION

These subroutines return the current values of the graphics parameters. Each subroutine requires an address in which to store the value to be returned. All of these subroutines force transmission of graphics data in the current buffer.

VI_QClip Returns the the current clipping rectangle.

VI_QColor Returns the current color of the screen: 0 means that bits having the binary value "0" will be black on the screen; 1 means that bits having the binary value "1" will be black on the screen.

VI_QDash Returns the current line dash pattern in the format described for *dash* (3G). If *dashlen* is 0, the lines are currently solid.

VI_QFont Returns the ID and name of the current font. The font ID is 0 if no font has been set. The pointer *fontname* should point to a block of characters large enough to hold a file name along with a string-termination byte. If you know beforehand the size of your file name, you may allow only as many bytes as required. Be aware of the string-terminator byte; there must be room for it.

VI_QMerge Returns the current merge mode in the format described for *merge*(3G).

VI_QPoint Returns the location of the current point. This command is especially useful after *string*(3G) has been issued, since character definitions can change the current point in unpredictable ways.

VI_QWidth Returns the current line width as a number between 1 and 16.

NOTE

VI_Query applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

clip(3G), color(3G), dash(3G), merge(3G), move(3G), string(3G), width(3G)

This page intentionally left blank.

NAME

VI_MRead, VI_FRead – read display data

SYNOPSIS

```
VI_MRead(x,y,wd,ht,data)
        int x,y;                    /* top left corner of area */
        int wd,ht;                  /* dimensions of area */
        unsigned short *data;       /* first byte of data */
VI_FRead(x,y,wd,ht,filename)
        int x,y;                    /* top left corner of area */
        int wd,ht;                  /* dimensions of area */
        char *filename;         /* name of file to place image in */
```

DESCRIPTION

These functions read display data into memory or into a file. The area to be read must be completely on the screen. The current setting of the clipping window is ignored.

VI_MRead Reads the specified area of the screen into the array passed as *data*. Image bytes are in the same format as expected by *MImage*. If the screen color is white, the bits are inverted on readback to make the data read back independent of screen color. The area to be read must be completely on the screen.

VI_FRead Reads the specified area of the screen and places it in the specified file. The file has the same format as expected by *FImage*. If the window color is white, data bits are inverted to make the data independent of the screen color.

NOTE

VI_Read applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

image(3G)

This page intentionally left blank.

NAME

VI_Run – process a log file

SYNOPSIS

```
int VI_Run(filename)
    char *filename;          /* log file name */
```

DESCRIPTION

VI_Run executes the commands logged in the specified file; *filename* is the name of a log file that was created by *VI_Login*. Using *VI_Run* with a log file has the same effect as executing *aedrunner*(1) from within a program, allowing a series of orders which require much calculation to be figured only once, logged, then quickly retrieved when needed.

NOTE

VI_Run applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

aedjournal(1), aedrunner(1), log(3G)

DIAGNOSTICS

VI_Run returns 0 for normal completion, and -1 if it detects any kind of inconsistency or unexplained results in the file.

This page intentionally left blank.

NAME

VI_String – draw a string

SYNOPSIS

```
VI_String(s)
    char *s;          /* string to draw */
```

DESCRIPTION

VI_String draws the specified string at the current point. Since a character definition is really a sequence of other graphics commands (usually *VI_MImage* and *VI_RMove),* the way in which characters are positioned, stepped, and drawn depends on the font definition. Character definitions typically modify the current point.

NOTE

VI_String applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

font(3G)

"Defining Fonts" in "The C Subroutine Interface for the IBM Academic Information Systems Experimental Display"

This page intentionally left blank.

NAME

VI_Tile – tile a rectangle

SYNOPSIS

```
VI_Tile(wd,ht,twd,tht,tile)
        int wd,ht;              /* dimensions of rectangle */
        int twd,tht;            /* dimensions of tile */
        unsigned short *tile;   /* first byte of pattern */
```

DESCRIPTION

VI_Tile fills a rectangle of the specified dimensions with the specified pattern. The rectangle's top left corner will be at the current point. The tile pattern must follow the rules for images as explained in *image*(3G), and can be of any size. The tile pattern is aligned to multiples of *twd* and *tht,* not to the bounds of the tiled rectangle, so that rectangular subareas of larger figures can be tiled without regard to their bounds, and the tile patterns will match. The current point is unchanged.

A full rectangle black or white fill can be most quickly drawn by requesting a one-by-one tile. Clearly, only all ON or all OFF may be drawn with this method, but any merge mode may be used.

NOTE

VI_Tile applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

image(3G)

This page intentionally left blank.

NAME

VI_Width − set line width

SYNOPSIS

```
VI_Width(width)
    int width;          /* line width */
```

DESCRIPTION

VI_Width specifies a value between 1 and 16 that is to be the line width. Line width is initially set to 1.

NOTE

VI_Width applies only to the IBM Academic Information Systems experimental display.

SEE ALSO

line(3G), query(3G)

This page intentionally left blank.

Appendix C. High C Programmer's Guide

This section contains the "High C Programmer's Guide," produced by MetaWare Incorporated.

This page intentionally left blank.

Appendix C. High C ™ Programmer's Guide

ABSTRACT

This is a guide to the operation of the High C compiler as implemented for Academic Information Systems 4.3 for the IBM RT PC ("4.3/RT"). It contains:

This page intentionally left blank.

1. INTRODUCTION

This is a guide to the operation of the High C compiler implemented for IBM Academic Information Systems 4.3 for the IBM RT PC ("4.3/RT").

The Compiler generates relocatable object modules directly, in contrast to most C compilers on UNIX operating systems, which generate assembly files.

High C was designed to facilitate serious professional programming. It is available on numerous processors. It supports the draft ANSI Standard (ANSI document X3J11/85-102, August, 1985) and a few extensions.

C is a mixed-level systems language designed by Dennis Ritchie at AT&T's Bell Laboratories. It grew in popularity because of its use in implementing the UNIX operating system, its elegant (and deceptive) simplicity, and its close-to-the-machine features. As its popularity has grown, many software developers have used it for real-world applications as well as systems software.

Later implementations of C were extended to add enumeration types and a few other features. More recently many extensions have been proposed to make C a safer language while still being consistent with the philosophy of the original language. Today there is a core language being standardized by the American National Standards Institute (ANSI).

High C includes what most likely will be ANSI Standard C and also provides extensions that were carefully designed to be consistent with the philosophy of C. Some of the best features of such other languages as Pascal, MetaWare's Professional Pascal, and Ada were incorporated as extensions. Incompatibilities were minimized by introducing a minimum of new key words and by retaining the original syntax. Yet the extensions are such that they will be flagged by any Standard-conforming compiler.

Portability. Standard C programs can be compiled with an ANSI option that turns off the extensions and reduces the language to the Standard core. Alternatively, such programs can be gradually upgraded by not choosing the ANSI option and using the extensions as required.

Safety, efficiency. While the close-to-the-machine features of C are available, High C supplies the new strong type-checking specified in ANSI C. In addition, the compiler provides many checking features usually available only in a separate "lint" program. Thus one gets both efficiency *and* reliability. It is an excellent language for both applications and systems programming.

Other important features and extensions include:

- three integer ranges and two floating-point precisions
- many compiler controls and options, including one for strict Standard checking
- nested functions complete with up-level references, as in Pascal
- nested functions passable as parameters to other functions, as in Pascal
- intrinsic functions, such as `_abs`, `_min`, `_max`, and `_fill_char`, for efficiency
- many optimizations, some of which are usually found only in mainframe compilers, including:

  common subexpression elimination
  retention and reuse of register contents
  dead-code elimination
  cross jumping (tail merging)
  jump-instruction size minimization
  constant folding
  numerous strength reductions
  automatic allocation of variables to registers

This guide contains all system-specific information necessary for using the compiler effectively. Readers new to the product should scan the Table of Contents for an overview of the guide. Briefly, we describe:

- how to compile, link, and run
- how to use compiler controls
- machine specifics, such as storage mapping and run-time organization
- defaults and limits
- communication with programs written in other languages
- listings and cross-references
- error messages

An extensive index provides for quick reference to all sections that discuss or significantly relate to each topic.

This guide does *not* explain the C language or the High C extensions. They are treated in the MetaWare *High C Language Reference Manual.* Neither this guide nor the manual attempts to teach C programming; consult the manual for references to several C textbooks.

2. INVOKING THE COMPILER

2.1 The hc Command

The **hc** command invokes the High C compiler, which translates C programs into executable load modules or into relocatable binary object modules suitable for linking with **ld**. The syntax of the command is:

hc *[options]... files...*

Any number of options and one or more files may be specified. Each option specified in the command applies to all the specified files for which it makes sense, except as noted below.

Several types of file names are allowed. A file name ending with ".c" is taken to be a C source module. It is compiled and an object module is produced with the same name as the source except with ".o" substituted for ".c". The ".o" file is normally deleted after linking when a single-module C program is compiled and linked.

A file name ending with ".s" is taken to be an assembly source module and is assembled, producing a ".o" file. Any other file specification is assumed to be an object module or archive library to be linked via **ld**.

All ".o" files are placed in the current working directory.

In general, **ld** is invoked if no compilation errors were detected and the **-c** option was not specified. The resultant load module is named "a.out" unless specified otherwise with the **-o** option (described below). Any argument beginning with a dash ("-") is taken as an option specification.

Example. The following command compiles the program in file sort.c, links it, and generates a load module named sort:

```
hc -o sort sort.c
```

2.2 Invoking the C Macro Preprocessor

The High C compiler has an integrated "inboard" macro preprocessor, documented in the *High C Language Reference Manual*. The preprocessor conforms to the proposed ANSI C Standard. However the "outboard" C macro preprocessor on most UNIX operating systems does not conform to the proposed Standard in some ways.

Because many C programs written for UNIX operating systems depend on minor idiosyncrasies of the outboard C preprocessor, the **-Hcpp/-Hnocpp**[1] options are provided. The **-Hcpp** option causes the outboard preprocessor to be invoked on the source file sending the output to a temporary file, which then serves as input to the compiler. **-Hnocpp** suppresses this action. The compiler is provided with the **-Hcpp** option on by default. The macro name __HIGHC__ is predefined, except when the **-Hansi** option is specified. The macro name __STDC__ is always predefined.

2.3 Command Options

Below is a description of each compiler option. Any option that is not recognized by **hc** is assumed to be a linker option and is passed on to **ld**. Options applicable only to High C are prefixed with an **H**.[1]

-Hansi Causes the compiler to accept only programs conforming to the proposed ANSI Standard.

Note: Since the proposed ANSI Standard is under revision at the time of this writing, this option's primary function is to turn off the High C language extensions.

-Hasm Directs the compiler to produce an assembly listing of the generated code on standard output, by initializing the Asm toggle to On. The assembly listing is annotated with lines from the main source file, but not with lines from any included files. These lines appear as comments immediately preceding the corresponding assembly instructions. If the **-S** option (described

---

1. "**H**" stands for High C. It is used to avoid conflicts with existing or future pcc options.

below) is also specified, the generated `.s` file is annotated with lines from the source file, and no listing is written on standard output.

-B*string* Finds substitute compiler executables in the files named *string* with the suffixes `cpp` and `hccom`. If no *string* is given, the default `/usr/c/o` is used; that is, the defaults are `/usr/c/ocpp` and `/usr/c/ohccom`.

-c Suppresses the invocation of **ld**, and forces an object file to be produced even if only one module is compiled.

-Hcpp Specifies that the outboard C macro preprocessor (`/lib/cpp`) is to be used, rather than the inboard preprocessor. **-Hcpp** is the default.

-Hnocpp Specifies the use of the inboard C macro preprocessor. See §2.2 *Invoking the C Macro Preprocessor* for details.

-D*name*
-D*name=def*

Defines the name *name* to the preprocessor as if by **#define**. If no *def* is given, the name is defined to be `1` (one). ***Note:*** There is no space between **-D** and *name*.

-E Specifies that the outboard C macro preprocessor is to be invoked and no compilation done. The preprocessor output is sent to standard output. **-E** overrides **-Hnocpp**.

-g Directs the compiler to emit additional symbol table information for the **dbx** debugger and omit certain optimizations..

Unless **-O** is specified, **-g** turns off the cross-jumping optimization and suppresses the deletion of unreferenced local functions.

-I*dir* Specifies an alternate directory to be searched to locate an include file. This option may be specified several times to indicate several directories to be searched. If a particular file is not located after searching the specified directories, one or more standard directories are searched. See §3 *COMPILER PRAGMAS*. ***Note***: There is no space between **-I** and the directory name *dir*.

-Hlines=*n* Causes a page eject to occur after every *n* lines written to standard output. The default of `60` is appropriate for most `6`-lines-per-inch printers, which have a total of `66` lines per page. The setting of lines is intended to allow some blank space at page boundaries. When using `8`-lines-per-inch, typically there are `88` lines per page, so **-Hlines** should be set to `80` or `82`. This option is used in conjunction with the **-Hlist** and **-Hasm** options. If *n* is `0`, no page ejects are emitted.

-Hlist Causes the compiler to generate a source listing on standard output. It works by initializing the `List` toggle to `On`. See §4 *COMPILER TOGGLES*.

-M Specifies that the outboard C macro preprocessor is to be invoked and `Makefile` dependencies are to be generated. The output is sent to standard output. No compilation occurs.

-m*x* Specifies a machine-dependent option. Currently available options are:

-ma Specifies that the C library function `alloca` may be called from within the source file(s). `alloca` must extend the stack frame of `alloca`'s caller and needs certain information about the size of the caller's stack frame. This option makes the information available in the caller's data area. If `alloca` is called from a function that was not compiled with the **-ma** option, an error diagnostic is generated at run-time.

-ms Causes the compiler to put out minimum-size floating-point data blocks (normally they are generously padded). This guarantees that the size of objects remains approxmimately that of previous releases, at the expense of performance.

-o *output* Is passed on to the **ld** command and names the final executable output file *output*. When this option is used, any existing `a.out` file is left undisturbed. ***Note***: White space *is* required after the **-o**.

-O Specifies that all optimizations supported by the compiler are to be performed on the generated code. This is the default unless **-g** is specified. Therefore, this option has meaning only when used in conjunction with **-g**.

-Hon=*toggle*
-Hoff=*toggle*
Turns a toggle `On` or `Off`. See §4 *COMPILER TOGGLES*.

-p Produces code that counts the number of times each function is called during execution. If **ld** is invoked, the profiling library `/usr/lib/libc_p.a` is searched in lieu of the standard C library `/lib/libc.a`. Also replaces the standard start-up function with one that automatically calls `monitor(3)` at the start and writes out a `mon.out` file. An execution profile can then be generated by use of `prof(1)`.

-pg Invokes a run-time recording mechanism as does **-p**, but keeps more extensive statistics and produces a `gmon.out` file. An execution profile can then be generated by use of `gprof(1)`.

-Hppo
-Hppo=*filename*
Specifies that the compiler is to invoke its inboard preprocessor only and send the results to *filename*. If **-Hppo** alone is given, the preprocessor output is printed to the standard output. No object module is generated, nor is **ld** invoked. "**ppo**" can be read "pre-process only" or "print preprocessor output". The preprocessor output is suitable for input to the compiler.

With **-Hppo**, any `Include` pragmas are *not* processed, since **-Hppo** turns off all processing past the preprocessor, and a later phase of the compiler handles the `Include` pragma. Alternatively, use **-Hon**=`Print_ppo` to obtain preprocessor output with processing of `Include` pragmas.

-R Makes all initialized static variables shared and read-only. This option is implemented by the assembler and therefore **-Hasm** acts as if `-S` was specified.

-S Produces an assembly source file instead of an object file (for each source file). The assembly source is written into a file with the same name as the C source with "`.c`" replaced by "`.s`". The file is always placed in the current working directory. No object file is written, nor is **ld** invoked.

Note: Unlike other compilers for UNIX operating systems, the High C compiler normally generates an object module directly, *without* producing an assembly file. The **-S** option essentially directs the last phase of the compiler to produce assembly source as the object code is generated. If the **-Hasm** option is also specified, the "`.s`" file is annotated with interlisted source file lines.

-U*name* Removes any initial definition of macro *name*. See **-D** above.

-v Causes the name of each subprocess to be printed as it begins to execute. (To get announcements of compiler-phase execution also, set **-Hoff**=`Quiet`.)

-Hvolatile
Forces the compiler to read from memory on all pointer dereferences. This is necessary only when pointers are used as addresses whose contents are "volatile" (can change via external forces).

-w Causes all warning messages from the compiler to be suppressed.

-H+w Issues *all* warnings, and comes highly recommended. The default is to issue only warnings that **pcc** would issue.

3. COMPILER PRAGMAS

The High C compiler provides "pragmas" (the term comes from Ada) that direct compiler operations. Pragmas control the inclusion and listing of source text, the production of object code files, the generation of optional additional program and debugging information, and so on.

3.1 Syntax of Pragmas

Compiler pragmas take one of the following general forms:

```
pragma <Pragma_name>;  /* or */
pragma <Pragma_name>(<Pragma_parameters>);
```

where `<Pragma_parameters>` is a list of constant expressions separated by commas. The number and types of the expressions are dependent upon the particular `<Pragma_name>`. A pragma can appear anywhere a statement or declaration can appear. See the *High C Language Reference Manual* for a specification of the precise placement of pragmas.

`<Pragma_name>`s are case insensitive.

3.2 Compiler Pragma Summaries

The following pragmas are available:

Pragma	*Purpose*
	Toggles — see §4 *COMPILER TOGGLES*:
`On,Off,Pop`	Turns `On` or `Off`, or reinstates a prior status of a compiler switch or "toggle".
	Externals — see §8 *EXTERNALS*:
`Alias`	Specifies the external name to be associated with a global identifier.
`Data`	Specifies the use of named blocks for data storage allocation. This is primarily intended for communicating with other languages.
	Inclusions — see §3.3 *COMPILER PRAGMAS/***`Include`** *Pragmas: Including Source Files*:
`Include`	Includes the source of another file in the compilation unit.
`C_include`	Conditional form of `Include`.
`R_include`	Includes the source of another file in the compilation unit, treating the path name as Relative to the directory of the file containing the pragma. This pragma treats the path name in the same manner as the **`#include`** preprocessor directive.
`RC_include`	Conditional form of `R_include`.
	Listings — see §10 *LISTINGS*:
`Page`	Causes page ejects to be inserted into the listing. This pragma takes effect only when the `List` toggle is `On`.
`Skip`	Causes blank lines to be inserted into the listing. This pragma takes effect only when the `List` toggle is `On`.
`Title`	Causes a title to appear at the top of each subsequent page. This pragma takes effect only when the `List` toggle is `On`.

3.3 `Include` Pragmas: Including Source Files

When source text from an alternate file is to be included in a compilation, the **`#include`** preprocessor directive is commonly used. The High C compiler supports, in addition, pragmas with alternate search strategies for including files. This section describes the various strategies used to search for include files.

Note: `Include` pragmas are processed only by the High C compiler. If an outboard preprocessor is to be used, we recommend using the **`#include`** directive rather than the `Include` pragma, as the outboard preprocessor will not process files included via the `Include` pragma. Thus the command line option **`-Hocpp`** should be specified when the `Include` pragma is used. See §2 *INVOKING THE COMPILER*.

The `Include` pragma is used to include source from other files while the compilation unit is being compiled. The pragma operates slightly differently from the standard C **`#include`** directive. There are four forms of the `Include` pragma:

```
pragma    Include(<File_name>);
pragma  C_include(<File_name>);
pragma  R_include(<File_name>);
pragma RC_include(<File_name>);
```

where `<File_name>` is a string *constant* denoting the name of a file.

Examples:

```
pragma   Include("a_lot");
pragma R_include("dclns");
pragmaC_include("math.h");
```

The `Include` pragma directs the compiler to include a file unconditionally. The `C_include` pragma causes the file to be included only if it has not been included before — "conditionally included". `R_include` has exactly the same effect as the standard C **`#include`** directive; that is, it is a "relative include". `RC_include` does a "conditional relative include".

The term *relative include* refers to an include in which the file is first sought relative to the directory of the file where the include pragma appears. If the file is not found in that directory, then any directories specified in any **`-I`** command line options are searched in order of appearance. See §2 *INVOKING THE COMPILER* for a description of the **`-I`** option. If the file is still not found, then one or more standard directories are searched.

A *non-relative include* refers to an include in which the file is first sought relative to the current working directory irrespective of the location of the file in which the `Include` pragma appears. If the file is not found in that directory, then any directories specified in any **`-I`** command line options are searched in order of appearance. See §2 *INVOKING THE COMPILER* for a description of the **`-I`** option. If the file is still not found, then one or more standard directories are searched.

A file name specification that begins with "/" is assumed to be an absolute file reference and no directories are searched.

Preprocessor directive **`#include`** "*`filename`*" specifies a relative include.

Directive **`#include`** <*`filename`*> specifies that only the **`-I`** and standard directories are searched.

Warning. There should be nothing to the right of an `Include` pragma. After the `Included` file is processed, processing resumes on the line immediately following the one containing the `Include` pragma. In effect, the rest of the line is a comment.

Identity of file names. For the `C_include` and `RC_include` pragmas, file names, including path, are considered the same only if they are textually identical. Thus, these two pragmas may cause two includes to occur:

```
pragma C_include(              "strings.h");
pragma C_include("/usr/include/strings.h");
```

even though both includes may refer to the same file.

Also, for the purposes of textual comparison, file name casing *is* significant, due to the operating system casing convention.

Methodology. The primary use for conditional includes is to support modularity.

Assume file `trees.h` is merely a collection of declarations defining the interface to a trees module. Suppose further that `trees.h` makes reference to a type `Symbol` in another module defined in `symbols.h`. If a standard **`#include`** `"symbols.h"` were placed within `trees.h`, a duplicate declaration of `Symbol` would occur in any compilation unit that `Included` both `trees.h` and `symbols.h`. If, instead, a conditional include were used in both `trees.h` and any compilation unit including `symbols.h`, at most one copy of `symbols.h` would be included.

With conditional includes, each interface file `F` can conditionally include all other interface files that are necessary for the definition of the resources in `F`. Therefore any user of `F` can simply `Include F` and automatically gets other resources that are needed, without duplication.

4. COMPILER TOGGLES

Pragmas can be used to turn `On` and `Off` various compiler switches or "toggles". In such cases, the pragma syntax is:

```
pragma <Pragma_name>(<Pragma_parameter>);
```

The `<Pragma_name>` is either `On`, `Off`, or `Pop`, and the single `<Pragma_parameter>` is the name of the toggle to be affected. All compiler toggles are described below.

`On` turns the toggle on; `Off` turns it off; and `Pop` reinstates it to a prior value. Toggles operate in a stack-like fashion, where each `On` or `Off` is a "push" of **on** or **off**, and a `Pop` "pops" the stack. The stack for each toggle is at least 16 elements deep, but no diagnostic is given if the stack overflows or underflows. ***Examples:***

```
pragma On (List);    -- Turns on the source listing.
pragma Off(List);    -- Turns off the source listing.
pragma On (List);    -- Turns on the source listing.
pragma Pop(List);    -- Back to off for the listing.
pragma Pop(List);    -- Back to on  for the listing.
```

Recall that toggles can also be initialized on the command line, with **-Hon** and **-Hoff**. See §2 *INVOKING THE COMPILER.*

The default values, names, and meanings of the compiler toggles are described below.

`Align_members` — Default: On

When `On`, causes members of structures to be aligned. When `Off`, no such alignment takes place. See §5 *STORAGE MAPPING.*

`Asm` — Default: Off

When `On`, causes an assembly listing to be generated, annotated with source code as assembly comments. If the `Asm` toggle is to be turned `On` and `Off` over sections of a module, the pragma should appear among executable statements rather than declarations for best results; otherwise, the point at which the pragma takes effect may not be obvious.

`Auto_reg_alloc` — Default: On

When `On`, causes the compiler to allocate **auto** variables to registers automatically. The compiler weights variables used within loops more heavily than those not so used in making its decision which variables to allocate to registers; furthermore it does not allocate to registers any variables that are used too infrequently. See §5 *STORAGE MAPPING.* A call of `setjmp` or `_setjmp` disables `Auto_reg_alloc` for the entire containing function.

`Char_default_unsigned` — Default: On

When `On`, causes type **char** to be unsigned by default.

The Standard allows the type **char** by itself, that is, without the adjectives **unsigned** or **signed**, to be either signed or unsigned. Of course, the types **unsigned char** and **signed char** can be used to explicitly control signedness.

`Double_return` — Default: On

When `On`, causes any function returning type `float` to instead return type `double`. For certain numeric coprocessors, such as the Motorola 68881 or Intel 80x87, this is of little consequence since the coprocessor already uses `long double` math exclusively. However, other coprocessors, such as the Weitek 1167, use both single and double formats internally. A program that uses `float`s predominantly would incur extra overhead were `float`-returning functions changed to return `double`; hence this toggle.

The toggle applies to any functions declared within the range in which the toggle is on. Functions declared with the toggle `Off` instead suffer the conversion.

Downshift_file_names — Default: Off

When `On`, causes the file name specification of any subsequent `Include` pragma to be interpreted as if it were in all lower case. This toggle is useful when moving source code from an operating system in which file name casing is not significant to a system in which it is significant.

Emit_line_table — Default: Off

When `On`, causes the compiler to add entries to the symbol table that associate source line numbers with object code addresses. Debuggers use this information to associate object code with source lines.

The **-g** command-line option turns this toggle `On`.

Note: This toggle does not affect the size of the generated code, but it does add about eight bytes per statement to the object module's name list.

Int_function_warnings — Default: Off

When `Off`, suppresses the warning message normally generated when a function returning **int** has no **return** `exprn;` statement within it, or a function returning **int** contains a **return;** within it.

This is to remove frequent warnings for old C source that did not use the reserved word **void** to indicate a function returning no result, because such functions return **int** by default.

List — Default: Off

When `On`, causes the compiler to produce a listing on standard output. It is typically given when starting the compilation but may appear in the source file to turn the listing `On` or `Off` around a particular section of source.

Literals_in_code — Default: On

When `On`, causes lengthy literals in a program to be placed in the code space rather than in the data space.

Note: Not all C literals can be placed in code. A string literal is a writable data item and hence cannot be placed in code; for such a literal `Literals_in_code` has no effect. See `Read_only_strings` below.

Long_enums — Default: On

When `On`, causes any variable of an **enum** type to be mapped to a fullword so as to be compatible with the portable C compiler `pcc`.

Make_externs_global — Default: On

When `On`, any local declaration of an object with storage class **extern** is made global if there is not already a global declaration of the object. Early C compilers promoted an **extern** declaration within a function to the global scope. This toggle supports programs depending upon that "feature".

Optimize_for_space — Default: Off

When `On`, causes the generation of more space-efficient but potentially less time-efficient code. May have no effect for some machines[2].

Optimize_xjmp — Default:On

When `On`, enables the cross-jumping optimization. This is an effective space-saving optimization that leaves execution time invariant. It slows the code generator slightly, and can produce code that is difficult to debug. See §A *CROSS-JUMPING OPTIMIZATIONS* of this guide for more information on the specifics of this optimization. See also the `Optimize_xjmp_space` toggle below[2].

Optimize_xjmp_space — Default: On

When `On`, enables a cross-jumping optimization that saves space, but always at the expense of time. This toggle takes effect only if `Optimize_xjmp` is also `On`. This optimization slows the code generator slightly, and can produce code that is difficult to debug. See §A *CROSS-JUMPING OPTIMIZATIONS* of this guide for more information on the specifics of this optimization. See also the `Optimize_xjmp` toggle above[2].

---

2. It is not advisable to use optimizations in a debugging/emulation environment.

Parm_warnings — Default: On

When On, causes the compiler to produce warnings whenever arguments to a non-prototype (old-style) function F do not match the types of the declared formal parameters of F. Frequently this inconsistency is a source of disastrous or difficult-to-find bugs.

Example:

```
double square(x) double x; {return x*x;}
...
printf("%d\n",square(3));
```

square is passed the integer 3, not the double 3.0, and the compiler issues a warning. The C language definition *prohibits* the compiler from casting 3 to a **double** before passing it.

To eliminate the compiler warnings, turn Off the toggle Parm_warnings. We recommend, however, that the program text be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

PCC_msgs — Default: On

When On, the diagnostic capabilities of the compiler are reduced to the **pcc** ("portable C compiler") level, in that the following warnings are not emitted:

```
Function called but not defined.
Function return value never specified within
   function.
This "return" should return a value of type ttt
   since the enclosing function returns this type.
"=" used where "==" may have been intended.
Only fields of type "unsigned int" or
   "unsigned long int" are supported.
External function is never referenced.
Declared type is never referenced.
```

The next four messages are suppressed for *global* variables when PCC_msgs is On:

```
Variable is never used.
Variable is referenced before it is set.
Variable is referenced but is never set.
Variable is set but is never referenced.
```

When all warnings are enabled in High C, code must be "squeaky clean" to get through the compiler without a warning. Some users have code that was designed with a compiler that is not so demanding, and would prefer fewer prods from the compiler. Hence the PCC_msgs toggle is supplied.

Pointers_compatible — Default: Off

When On, allows pointers of any type to be compatible with each other. Although this is in violation of the Standard and High C specifications, many old C programs improperly assign pointers of different types to each other. This toggle allows such programs to be compiled without modification.

Pointers_compatible_with_ints — Default: Off

When On, allows pointers of any type to be compatible with **int**s. Although this is in violation of the Standard and High C specifications, many old C programs improperly assign pointers to **int**s and vice-versa. This toggle allows such programs to be compiled without modification.

ANSI and High C disallow this dangerous practice because pointers are not necessarily the same size as **int**s on all machines. The programmer should ensure that intermixed pointer and **int** values have the same size; otherwise a pointer stored in an **int** may not be retrieved as expected.

Print_ppo — Default: Off

When On, causes preprocessed input to be written to standard output. With this toggle, it is possible to print what the compiler proper receives over a local area of source code. This toggle is used to inspect the expansion of a macro, by turning the toggle On prior to the macro invocation and Off after it. *Note*: This toggle is ignored unless **-Hnocpp** is specified or is the default.

Print_protos — Default: Off

When On, causes the compiler to write to standard output a new, prototype-style function header for each function definition. This toggle aids in the conversion of C programs to the ANSI prototype syntax derived from the C++ language. For example, for the function definition:

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces:

```
int f(int *x, double (*y)(), int *z);
```

The old function header can then be replaced with the generated one.

There is a minor pitfall in having the compiler automatically generate prototype headers: array parameters, according to the semantics of C, are converted to pointer parameters.

Print_reg_vars — Default: Off

When On, causes the compiler to report (on standard output) each variable that is mapped to a register. This saves the programmer the trouble of looking at the generated code to discover such information.

Prototype_conversion_warn — Default: On

When On, causes the compiler to generate a warning message when a function's argument is converted due to a prototype declaration.

When using function prototypes, the compiler may automatically convert a function's argument so that the argument's type matches that of the formal parameter. Wherever such a conversion does not match what would happen in the *absence* of prototypes, such C code would probably not run correctly on older C compilers that lack prototypes. Turn On toggle Prototype_conversion_warn to have the compiler flag all such occurrences.

Prototype_override_warnings — Default: On

When On, causes the compiler to produce a warning whenever a declaration (not definition) for a function using the new prototype syntax overrides the semantics of an old-style function definition.

Standard C requires that function prototype declarations override old-style function definitions. This means that the simple inclusion of a .h header file with prototype declarations of functions obtains the new prototype semantics for the definitions of those functions. This feature has both disadvantages and advantages.[3]

The advantage is that the new prototype semantics — the Pascal-style assignment-conversion of arguments to the types of the formal parameters — is obtainable by merely including a declaration in a header file. The disadvantage is that a definition can no longer be read out-of-context; without searching header files one cannot determine whether the compiler compiles the function using prototype-style semantics or not. For example:

file header.h:

```
int func(float f,long l);
```

file prog.c:

```
#include "header.h"
int func(f,l) float f; long l; {
   ...
   }
void sub() {
   func(3, 4.4);    /* Passes 3.0 and 4L via */
   }                /* automatic conversion. */
```

Were header.h *not* included, the call to func in sub would pass the **int** 3 and the **double** 4.4; func would probably not work right. With the header file included, the interface for func is changed to

---

3. The interested reader may wish to consult the Winter 1987 (volume 2, number 3) issue of the **C Journal** for an article by Tom Pennello of MetaWare on the subject of prototypes.

prototype-style (`3` is converted to **`float`** and `4.4` to **`long`**). Thus, one can know how the compiler treats `func` only by searching all of the header files.

To obviate the need for searching, High C provides a warning message whenever an old-style definition is overridden by a prototype. The warning message can be disabled by turning `Off` toggle `Prototype_override_warnings`.

We recommend that function definitions be written with the new prototype syntax for improved readability and reliability. To wit:

file `prog.c`:

```
int func(float f,long l) {
    ...
    }
```

The ANSI committee permitted the override feature for two reasons: first, it would take some work to convert programs to use the new syntax in the definition (although with toggle `Print_protos`, High C generates the headers from old-style definitions); second, most compilers do *not* support prototype-form definitions, and the use of a header that is conditionally included based upon the compiler being used makes code more easily compilable by different compilers.

`Public_var_warnings — Default: On`

When `Off`, suppresses the warning messages:

```
Variable is never used.
Variable is referenced before it is set.
Variable is referenced but is never set.
Variable is set but is never referenced.
```

for all variables exported, that is, non-automatic variables not declared **`static`** or **`extern`**.

Such messages occur only for such variables that are not declared within a **`#included`** file. If one adheres to the discipline that all imported variables are defined in included files, the message does not occur.

`Quiet — Default: On`

When `Off`, causes each compilation phase to be announced in turn as the compilation progresses. (This toggle is not turned `Off` by **`-v`**.)

`Read_only_strings — Default: Off`

When `On`, string literals are considered true literals. Identical string literals appear in the object code only once and the `Literals_in_code` toggle (see above) takes effect for string literals, causing them to be placed in code.

C string literals are not true literals since they are writable data items. This means that they cannot normally be placed in code space. Furthermore, two identical C string literals must normally be duplicated in a program's object code, since one might be modified and the other not. To avoid this, use `Read_only_strings` and `Literals_in_code`. These two toggles cause C string literals to be placed in code.

The **`-R`** option turns `Read_only_strings On` initially.

`Summarize — Default: Off`

When `On`, causes the production of summaries of compilation activities. The summaries are produced at various stages of compilation.

`Warn — Default: On`

When `Off`, causes warning messages to be suppressed. The **`-w`** option turns `Warn Off` initially.

5. STORAGE MAPPING

5.1 Data Types in Storage

The table below summarizes the size and alignment of various C data types, and whether a variable of each type can be allocated to a register.

char and **int** types have the same size regardless of whether they are signed; therefore the table does not mention the sign.

Data Type	*Size*	*Alignment*	*Allocable*
char	1 (bytes)	1 (bytes)	Y
short int	2	2	Y
int	4	4	Y
long int	4	4	Y
float	4	4	Y
double	8	4	Y
long double	8	4	Y
enum	. . . See Note 3 . . .		
Pointer	4	4	Y
Full-function[4]	8	4	N
T[n]	n***sizeof**(T)	Same as T	N
struct{…}	See Note 1.	See Note 2.	N
union {…}	See Note 1.	See Note 2.	N

Note 1: The size of a **struct** or **union** is dependent upon whether the compiler generates padding to align members. The compiler will generate such padding by default if the toggle `Align_members` is `On`, and will not do so by default if the toggle is `Off`. The keywords **_packed** and **_unpacked** have been added to High C to allow control over member alignment on an individual **struct** or **union** basis. A **_packed struct** is not padded; an **_unpacked struct** is padded. See §4 *COMPILER TOGGLES* to determine the default setting of `Align_members`.

The size of an unpadded **union** is the size of the biggest member. The size of a padded **union** is the size of the biggest member padded so that its size is evenly divisible by its alignment.

The size of an unpadded **struct** is the sum of the sizes of its members. Non-bit-field members always start on byte boundaries, and there is no padding between members except in the case of bit fields; see below. The size of a padded **struct** is the sum of the sizes of its members including alignment padding between members. It is padded so that its size is evenly divisible by its alignment.

Note 2: A **struct** or **union** is aligned according to the most stringent requirements among its members.

Note 3: The size of **enum** types depends on the status of the `Long_enums` toggle. If the toggle is `Off`, the type is mapped to the smallest of a byte, half-word, or full word, such that all the values can be represented. If the toggle is `On`, the **enum** maps to a full word (matching the convention of the Portable C Compiler). See §4 *COMPILER TOGGLES.*

Bit members. Only unsigned bit members are supported. A bit member may not exceed 32 bits and is packed in each consecutive byte as shown in the map below. A bit member must fit within a four-byte word that is aligned to a four-byte boundary. Padding is added where appropriate to make this true.

A bit member of length zero causes alignment to occur at the next full-word boundary, that is, where an **int** would be aligned.

For example, the structure definition:

```
struct {unsigned x:11,y:9,z:13,w:1; char c; short i;}
```

---

4. A full-function value is a High C extension. It consists of a function address and a static link. See the *High C Language Reference Manual* for details.

is mapped to memory as follows:

```
7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
<---Byte 0 --->  <---Byte 1 --->  <---Byte 2 --->  <---Byte 3 --->
x x x x x x x x  x x x y y y y y  y y y y

7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
<---Byte 4 --->  <---Byte 5 --->  <---Byte 6 --->  <---Byte 7 --->
z z z z z z z z  z z z z z w      c c c c c c c c

7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
<---Byte 8 --->  <---Byte 9 --->
i i i i i i i i  i i i i i i i i
```

5.2 Storage Classes

Each **static** variable is placed in either the BSS section or the DATA section — the latter if it is initialized.

Each global variable with no **extern** specifier that is not initialized is defined as a common block; if it is initialized, it is mapped into the DATA section and given the *global* attribute. Each **extern** variable is given the *global* and *undefined* attributes.

Each **auto** variable is assigned either to a machine register or to storage in the routine's "stack frame". See §6 *RUN-TIME ORGANIZATION*. The compiler chooses which of the **auto**-classed variables to place in registers based upon the variable's type, frequency of reference, and whether the & operator is ever applied to it. In a function containing calls to `setjump`, **auto** variables are not mapped to registers, so that their values are not lost across such calls.

Each **register** variable is assigned similarly, except that it is given extra weight in assignment to a machine register. ***Be warned that*** use of library functions `setjmp` and `longjmp` can produce unpredictable results in the context of **register** variables. See *setjmp(3)*.

6. RUN-TIME ORGANIZATION

The High C compiler adheres to the standard linkage convention established by 4.3/RT[5]. This chapter presumes knowledge of the 4.3/RT architecture and assembly language. Throughout this chapter the term *word* denotes a four-byte storage unit.

6.1 Register Usage

Certain registers, such as `r1`, have specific uses throughout execution; others, such as `r15`, are used during a function call and are free at other times. The following table defines register usage at the call interface.

Register	Saved over call	Use
`r0`	no	called-function data area pointer
`r1`	yes	stack pointer (caller's frame pointer)
`r2`	no	argument word 1 and returned value
`r3`	no	argument word 2 and lower half of a returned double value
`r4`	no	argument word 3
`r5`	no	argument word 4
`r6-r12`	yes	register variables, etc.
`r13`	yes	frame pointer
`r14`	yes	data area pointer
`r15`	no	return address
`mq`	no	multiply/divide register

In addition, floating-point registers 0 and 1 are not saved over a call; registers 2-7 are preserved.

6.2 The Data Area

Each C function has an "entry point" and a "data area". Both must be referenced at the point of a call.

The *entry point* is where the code of the function begins. The *data area* (also called a "constant pool", which is a misnomer) contains strings, function addresses, and other literals.

A function `foo` normally has an entry point named `_.foo` and a data area named `_foo`.

The call instruction sequence sets `r0` to the address of the called-function's data area. The first word in the data area is the entry point of the called function. The word following supports the code profiling option (**-p**), and if present must be initialized to zero; the third word, also optional, supports `alloca` storage allocation.

In the function prologue code, `r0` is copied to `r14`; thus, `r14` is used to address the associated data area from within a function.

The data area is placed in the `DATA` section so that `r14` may be used as a base for referencing local static variables. Static variables are usually mapped before the various data areas; therefore, static variable references employ negative offsets from `r14`.

When a pointer to a function is assigned the "value" of a function, it is actually assigned the address of the function's data area. The first word of the data area always contains the entry point, that is, the address of the first instruction of the function.

---

5. Portions of this chapter copyright International Business Machines Corporation, 1987. Excerpts by permission, from the manual entitled *Academic Information Systems 4.3 for the IBM RT PC*. More information may be found in the section entitled "4.3/RT Linkage Convention" in Part II, Supplementary Documents.

6.3 Stack Frame Layout

The stack holds frames for currently active functions. It is word-aligned and grows downward. `r1`, the "stack pointer", indicates the low address of the stack frame of the currently executing function.

A stack frame is divided into the following areas, highest address first:

a) space for incoming argument list (four words)
b) linkage area (four words; reserved)
c) static link (one word)
d) general register save area (sixteen words maximum)
e) floating-point register save area (zero or eighteen words)
f) local variables and temporary storage
g) words 5 through n of out-going argument lists

The *static link* applies to a function that is nested within another function; it is the address of the enclosing function's stack frame. (Nested function definitions, as in Pascal, are a High C extension to Standard C.) The static link is used to do "up-level addressing", that is, referencing local variables of containing functions. While executing level-one functions, the static link field is uninitialized.

The caller's return address (`r15`) is saved at a fixed offset of ten words below the top of the stack frame, at the top of the general register save area.

The floating-point register save area is up to eighteen words long. It is empty if no such registers need preserving.

The compiler uses `r13` to reference the top of the stack frame. Since it is more efficient to access variables with small positive displacements, the compiler often biases the value of `r13` to improve the code for local variable accesses (see §6.7 *Prologue* below for more information).

6.4 Argument Passing

Arguments are word-aligned and allocated to consecutive words on the stack. The list lies across frame boundaries: words 1-4 are allocated in the top of the callee's frame, and the remainder are in the bottom of the caller's frame, which is adjacent. In a call, words 1-4 are actually passed in registers `r2-r5`.

Arguments are passed as follows, based on argument type:

- An **`int`** is passed in a single word.
- A **`long`**, **`short`**, pointer, or **`char`** is treated as an **`int`** and passed in a word.
- A **`double`** is passed in two consecutive words.
- A **`float`** is converted to **`double`** and passed in two consecutive words, unless it is being passed to a prototyped function that was declared to receive a **`float`**, in which case it is passed in a word.
- A structure is aligned to a word and left justified, except for a structure one, two, or three bytes long, which is right justified.
- A pointer to a function is passed as a pointer to the function's data area.
- A full-function value[6] is passed as two words. The first contains the address of the data area; the second contains the static link.

If a function is declared as returning a structure, the caller passes the address of a result area in `r2`. The first word of the explicit argument list is passed in `r3`. Subsequent arguments are shifted accordingly.

---

6. A full-function value is a High C extension. It consists of a function address and a static link. See the *High C Language Reference Manual* for details.

6.5 Function Results

A result is returned from a function in one of three ways, depending on the function's return type:

- an **int**, **long**, **short**, pointer or **char** is returned in r2.
- a **double** is returned in r2 and r3.
- a **float** is widened and returned as a **double**.
- a structure result or full-function value is returned by moving it into the area pointed to by the first word in the argument list (in r2 on entry).

6.6 Calling Sequences

A call of a known function foo first prepares the argument list, then executes the following:

```
balix r15,_.foo          # Call.
l     r0,$.long(_foo)    # Get its data area
                         # pointer, r14 relative.
```

If the function being called is nested within another function (High C, not plain C), the caller stores the static link, that is, the frame pointer of the enclosing function, into -36(r1) before executing the **balix**.

Note that the address of the data area of the function being called is in the data area of the caller and is referenced off of r14.

A call to a function via a function pointer is done as follows. Recall that a function pointer addresses the function's data area. If the pointer is in r8, typical code is:

```
ls    rt,0(r8)           # Get address of entry point.
balrx r15,rt             # Call.
mr    r0,r8              # Load r0 with data area address.
```

6.7 Prologue

Prologue code saves the caller's registers, establishes the frame pointer (r13), and obtains stack space for the stack frame. Typical code is:

```
_.foo:  stm   rn,-76+(n-6)*4(r1)# Save caller's regs.
        mr    r14,r0            # Set up addressability
                                #   to data area.
        mr    r13,r1            # Set up frame pointer.
        cal   r1,frame_size(r1) # Allocate stack frame.
```

Here *n* (6≤*n*≤13) is the register number of the first general register to be saved, and *frame_size* is the size of the stack frame (word-aligned) including the space required for the caller's save area. Other instruction sequences are needed for frame sizes larger than 32,767 bytes.

If floating-point registers must be saved, the following code is inserted before the allocation of the stack frame:

```
cal   r15,stm(r14)
balr  r15,r15
```

where stm(r14) references a floating-point **storem** instruction to save non-volatile floating-point registers in the floating-point save area. See the section entitled "4.3/RT Linkage Convention" in Part II, Supplementary Documents.

As noted earlier, r13 may be biased by some negative amount so as to improve code references to stack frame variables. For example, "**mr** r13,r1" may be replaced with "**cal** r13,-80(r1)".

6.8 Epilogue

The epilogue restores the caller's environment and returns control. Typical code is:

```
mr    r1,r13                 # Restore stack pointer.
lm    rn,-76+(n-6)*4(r1)     # Restore general reg.
br    r15                    # Return to caller.
```

where *n* is the same value as in the **stm** instruction of the corresponding prologue.

If floating-point registers are involved, these instructions appear before the **lm** instruction:

```
cal   r15,lm(r14)
balr  r15,r15
```

where `lm(r14)` references a floating-point **loadm** instruction to restore those floating-point registers saved in the prologue.

6.9 Assembler Issues

Temporarily, all modules linked by **ld** must have the global symbol `.oVncs` defined as an absolute with value `0`. This distinguishes modules using an earlier linkage convention that is now obsolete. In assembly language, the symbol can be defined via:

```
.globl .oVncs
.set   .oVncs,0
```

The compiler also defines the following to help identify compilation specifics:

```
.globl .oVhcversion
.set   .VXxDy
```

where *version* indicates the compiler version number, such as `1.4`; x may be either `E` or `e`, meaning that the code was compiled with toggle `Long_enums` either `On` or `Off`, respectively; and y may be either `u` or `S`, meaning that the code was compiled with toggle `Char_default_unsigned` either `On` or `Off`, respectively.

7. SYSTEM SPECIFICS

This section describes some system-specific aspects of the High C compiler for IBM Academic Information Systems 4.3 for the IBM RT PC.

7.1 Floating-Point Arithmetic

High C uses the IEEE Standard 754 formats to represent floating-point data.

Each **float** is a 32-bit value with an 8-bit exponent and a 23-bit mantissa. The absolute values of the representable numbers lie in the range 8.43×10^{-37} to $3.37 \times 10^{+38}$.

Each **double** and **long double** is a 64-bit value with an 11-bit exponent and a 52-bit mantissa. The absolute values of the representable numbers lie in the range 4.19×10^{-307} to $1.67 \times 10^{+308}$.

7.2 Size of Complilation Unit

Each compilation unit is limited in size to perhaps 15,000 lines of "typical" C code, after macro expansion, due to a limit of 65K nodes in a tree representation of the entire module as expanded.

7.3 Some ANSI-Required Specifics

Here are some additional system specifics that the ANSI document X3J11/86-102 requests each C implementation provide.

Identifiers. The number of significant characters in an identifier is 32,000, since that is the longest input line acceptable to the compiler. Casing is preserved.

Characters. The characters in the source and the execution character set are the standard ASCII characters. Each character in the source character set maps into the identical character in the execution character set. Without exception, all character constants map into some value in the execution character set.

A character is stored in a byte and there are four bytes in an **int**.

High C does not permit a character constant that contains more than one character. Such a construction is usually machine-dependent.

The type specifier **char**, when not accompanied by an adjective, denotes an unsigned character type. However, this can be changed by turning `Off` the toggle `Char_default_unsigned`.

Integers. Integers are represented in twos-complement binary form. The following table illustrates the ranges of values to which the various integer types are restricted:

Type	*Range*		
signed char	-128	to	127
unsigned char	0	to	255
short	-32,768	to	32,767
unsigned short	0	to	65,535
int	-2,147,483,648	to	2,147,483,647
unsigned int	0	to	4,294,967,296
long	-2,147,483,648	to	2,147,483,647
unsigned long	0	to	4,294,967,296

Conversion of an integer to a shorter signed integer or **int** bit field is done by bit truncation; that is, when storing an `X`-bit value into a `Y`-bit receptacle, where `X` > `Y`, the rightmost `Y` bits of the first value are stored. Conversion of an unsigned integer `U` to a signed integer `I` where **sizeof**`(U)` = **sizeof**`(I)` consists in transferring the bits of `U` into `I`, whether or not the value of `U` is representable in `I`. For example, `(`**short int**`) (`**short unsigned**`) 65535` is the **short int** value `-1`. The **sizeof** operator returns an **int**.

The results of bitwise operations on signed integers are the same as if the integers were treated as unsigned.

The sign of the remainder on integer division is the same as the sign of the dividend.

The right shift of a signed integral type is arithmetic; that is, the sign bit is propagated to the right.

Floating point. Floating-point representation is IEEE Standard 754. The default rounding mode is "round to nearest". See §5 *STORAGE MAPPING* for the length required for each floating-point type.

When a negative floating-point number is truncated to an integral type, the truncation is toward zero. Thus `-2.7` is truncated to `-2` and `-1.2` to `-1`.

Arrays and Pointers. The type returned by **`sizeof`** is type **`int`**, and the difference of the pointers is type **`int`**.

Registers. A **`register`** variable is eligible for assignment to a machine register if its type is appropriate. See the table in §5 *STORAGE MAPPING* for a list of such types.

Potentially, as many variables can be placed in registers as there are "nonvolatile" registers. See §6 *RUN-TIME ORGANIZATION* for a list of the nonvolatile registers.

Structures, unions, and bit fields. Only unsigned bit fields are supported. A bit field declared as **`int`** is treated as **`unsigned int`**. For more information on structures, unions, and bit fields, see §5 *STORAGE MAPPING*.

Declarators. There may be at most 65,535 declarators modifying a basic type.

Statements. There may be at most 65,535 cases in a **`switch`** statement.

Preprocessing directives. A single-character constant in a constant expression controlling conditional inclusion is always non-negative in value, ranging from 0 to 255.

For the method of locating includable source files, see §2 *INVOKING THE COMPILER*.

8. EXTERNALS

The names of variables and functions that are communicated across module boundaries are normally made global in the resultant object module. In large programs there may be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately neither C nor most linkers provide for a structured name space — for named packages of resources, for example. Thus the well-chosen "internal" names in a program may not also be usable as "external" names (those known to the linker) as they should be. Thus some method of aliasing internal names to externals is needed, and High C provides it.

It is important to be able to alias such names to avoid conflicts in the linker's external symbol dictionary, rather than being forced to pervert the internal names themselves. It is the internal names that are most important to be well-chosen "containers of meaning", for program maintainability.[7]

8.1 The `Alias` Pragma

This pragma specifies, for a specific internal name, another name for external or public purposes. It is the alternate name that appears in the object module. The form of the `Alias` pragma is as follows:

```
pragma Alias(<Internal_name>,<External_name>);
```

where `<Internal_name>` is the function or variable identifier being aliased and `<External_name>` is a *constant* string expression whose value denotes the alternate or external name.

The `Alias` pragma must appear *in the scope of* the declaration of the internal name.
Example:

```
void Initialize();
   pragma Alias(Initialize,"x_initialize");
   /* The function Initialize is referenced in the */
   /* object-module name list as "x_initialize".   */
int BA;
   pragma Alias(BA,"PhD");
   /* "BA" is referenced in the name list as "PhD". */
```

8.2 Data Segmentation: the `Data` Pragma

Audience. This section may be skipped except by those interested in either (a) linking with programs written in Professional Pascal or (b) using a data communication convention different from that of Standard C.

Communication between separately-compiled modules is achieved by using the **`extern`** storage class in C. Multiple defining declarations of a variable `x` are allowed, as long as at most one of them initializes `x` (thus the **`extern`** storage class is not required).

The `Data` pragma provides an alternative method of sharing data, using named blocks. Its general usage is illustrated by:

```
pragma  Data(class,"blockname");
int      X,Y,Z;
...   /* Other normal C declarations may appear here. */
pragma  Data;
        /* "Turns off" the prior Data pragma. */
```

where *`class`* is one of `Common`, `Import`, or `Export`, and `"blockname"` is a constant string expression. The ending `Data` pragma has no parameters.

---

7. The external names are also important in that respect, but we believe that the proper solution is a "module interconnection language" and associated linker with a structured dictionary to match the overall structure of the program.

Only the given block name is made known to the linker as a global symbol: each variable is addressed at a fixed offset within the block. When the `Import` class is specified, the symbol is given the *undefined global* attributes and a value of `0`; when `Export`, the symbol is defined in the module's `bss` or `data` segment and given the *global* attribute. When `Common`, the symbol is flagged as a named common block, that is, given the *undefined global* attributes and a value that is equal to its length.

Scope. Each `Data` pragma must be terminated or "turned off" as illustrated above *in the same scope* in which it is turned on. The storage class specification applies only to variable declarations between the specification and its termination, *but not to any variables declared within embedded function definitions* (a High C extension). That is, variables declared at lower levels — local to surrounded (nested) function declarations — are not affected: at a function declaration, any active `Data` pragma temporarily becomes inactive and the default applies through the end of the function.

A compile-time warning is issued if a `Data` pragma is specified when a prior `Data` pragma is still active (in which case the subsequent pragma applies), or if a `Data` pragma is active at the end of a function declaration or at the end of a compilation unit. Thus `Data` pragmas cannot be nested within a single function, though they can be nested if they apply to the local variables of distinct functions.

Example:

```
pragma  Data(Common,"block");
int     Tables_are_loaded: Boolean;
struct {...} Tables;
pragma  Data;
```

Here, the names, `Tables` and `Tables_are_loaded`, are mapped at consecutive displacements (subject to boundary alignment) within the common block `block`.

9. ASSEMBLY LANGUAGE COMMUNICATION

9.1 Assembly Routines

§6 *RUN-TIME ORGANIZATION* describes the code that an assembly routine must execute to be callable from C, how arguments are passed, and how function results are returned. In short, an assembly routine should be coded according to the following guidelines. Symbols in *italics* are to be filled in appropriately.

```
        .text
        .globl  _.name
        .globl  _name
_.name: stm     rn,-76+(n-6)*4(r1)
        mr      r14,r0
        mr      r13,r1
        cal     r1,frame_size(r1)
#       The body of the routine goes here.
        mr      r1,r13
        lm      rn,-76+(n-6)*4(r1)
        br      r15
        .data
        .align  2
_name:
        .long   _.name
```

where *name* is the function's name as referenced from C; *n* ($6 \le n \le 13$) is the register number of the first general register to be saved; *frame_size* is the size of the stack frame (word-aligned) including the space required for the caller's save area.

9.2 Function Naming Conventions

An identifier that is global, that is, accessible across module boundaries, must have information provided to the linker that associates its name with its address. This is done by placing a corresponding name in the *name list* of the object module and giving it the "global" attribute.

There are two names associated with every function: one referring to the entry point and the other to the associated data area. The name that references the data area of a C function `foo` is `_foo`; the entry point is referenced by `_.foo`.

9.3 Examples: Calling Assembly from C

Example #1:

High C:

```
extern void and(int *dest, int *src, int len);
void main()
   {
   int a[256],b[256];
   ...
   and(a,b,256);
   ...
   }
```

Assembly:

```
        .data
        .globl  _and
        .globl  _.and
        .align  2
_and:   .long   _.and
        .text
_.and:
        stm     r13,-48(r1)
        mr      r14,r0
        mr      r13,r1
        cal     r1,-48(r1)
L:      cis     r4,0
        jle     exit
        ls      r0,0(r2)
        ls      r5,0(r3)
        n       r0,r5
        sis     r4,1
        bx      L
        sts     r0,0(r2)
exit:   mr      r1,r13
        lm      r13,-48(r1)
        br      r15
```

Since the assembly routine does not modify non-volatile registers and has a zero-length stack frame (except for the caller's save area), it can be optimized to the following:

```
        .data
        .globl  _and
        .globl  _.and
        .align  2
_and:   .long   _.and
        .text
_.and:
L:      cis     r4,0
        bler    r15
        ls      r0,0(r2)
        ls      r5,0(r3)
        n       r0,r5
        sis     r4,1
        bx      L
        sts     r0,0(r2)
```

However, if an exception should occur in the optimized routine, for example, an invalid address passed in, the debugger may be hampered in identifying the context.

Example #2:

High C:

```
extern char peek(char *adr);
void main(){
   char  b;
   ...
   b = peek(0x8000);
   ...
   }
```

Assembly:

```
        .data
        .globl  _peek
        .globl  _.peek
        .align  2
_peek:  .long   _.peek
        text
_.peek: lc      r2,0(r2)        # Return the byte.
        br      r15
```

9.4 Example: Calling C from Assembly

To call a C function `foo` from assembly language, first store the arguments in `r2` through `r5` (putting any additional arguments on the stack at `0(r1)`) and then execute the following two instructions.

```
        balix   r15,_.foo
        l       r0,x(r14)
```

where `x(r14)` refers to a memory location containing the address of `_foo`.

Example:

High C:

```
void write_string(char *s)
   {
   printf("%s\n",s);
   }
```

Assembly:

```
        .text
        .globl  _write_string
        .globl  _.write_string
_name:  .long   _.name
        .long   _write_string
_.name: stm     r13,-48(r1)
        mr      r13,r1
        mr      r14,r0      # Set up reference
        ...                 #   to data area.
        get     r2,$msg
        balix   r15,_.write_string
        l       r0,4(r14)   # i.e., _name + 4
        ...
msg:    .asciz  "This is a message."
```

9.5 Data Communication

A global variable "x" appears in the name list as "_x", unless specified otherwise with an `Alias` pragma — see §8 *EXTERNALS*.

§5 *STORAGE MAPPING* explains how the various C data types are mapped into storage. Note that uninitialized global variables without the **extern** qualifier are actually defined as individual common blocks. The following examples illustrate the sharing of variables across C and assembly modules:

High C:

```
int alpha,beta;
char hextable[] = "0123456789ABCDEF";
extern  char *names[]; /*A read-only table of names.*/
extern  short status;
```

Assembly:

```
        .comm   _alpha,4
        .comm   _beta,4
        .globl  _hextable  # Imported from C.
        .text
        .globl  _names     # Read-only;
_names: .long   L01        # in text segment.
        .long   L02
        .long   L03
        .long   0
L01:    .asciz  "alfred"
L02:    .asciz  "bonny"
L03:    .asciz  "charlie"

        .data
        .globl  _status
_status:
        .short  0
```

High C provides the ability to map more than one variable into a named block, for example, a common block as in FORTRAN. This facility is provided by the `Data` pragma and is documented in §8 *EXTERNALS*. The following illustrates how such a common block may be accessed from assembly language.

C Common Block Definition:

```
pragma Data(Common,"BLOCK_NAME");
int    a,b;
char   c,d;
short  e;
pragma Data;
```

Assembly Language Equivalent:

```
        .comm   BLOCK_NAME,12
        .set    a,0
        .set    b,4
        .set    c,8
        .set    d,9
        .set    e,10
        ...
Usage:
        get     r2,BLOCK_NAME
        l       r3,a(r2)    # Load value of a.
        l       r4,b(r2)    # Load value of b.
        lc      r5,c(r2)    # Load value of c, etc.
```

Note that variables `a,b,c,d,` and `e` are *not* global; that is, they do not appear in the name list with the "global" attribute. The only name that appears in the name list is `BLOCK_NAME`.

10. LISTINGS

This chapter describes the format of a listing generated by the compiler. Those pragmas that have an effect on the listing are described as well.

10.1 Pragmas `Page, Skip, Title`

To cause `n` page ejects at some point in the listing, insert:

```
pragma Page(n); /* where n is the number of ejects. */
```

To cause `n` lines to be blank at some point in the listing, insert:

```
pragma Skip(n); /* where n is the number of blanks. */
```

To cause a title `T` to appear at the top of each successive page, place the following pragma in the source:

```
pragma Title(T); /* where T is a string constant. */
```

Each successive `Title` pragma changes the title for subsequent pages; therefore the title does not appear on the first page.

10.2 Format of Listings

Ruler. The first line after any header and title lines on each page is a "ruler" that defines three fields for each line. The fields are for: (1) three level numbers, (2) the line number, and (3) the line contents. The ruler is as follows:

```
Levels LINE#|----+----1----+----2----+----3----+----4----+----5--...
```

Level-numbers can be used to find a missing `}` or comment terminator when a message such as "`Unexpected end-of-file.`" is produced by the compiler. All three level-numbers are initially zero, but they are printed as blank rather than `0`.

The first level-number indicates the scope nesting level for **`struct`** or **`union`** declarations.

The second level-number indicates the statement nesting level. It is incremented at the beginning of each `{` and is decremented at the corresponding `}`.

The third level-number indicates the structure initialization nesting level. It is incremented at the beginning of each `{` and decremented at the corresponding `}`.

Include files. A first-level `Include` file named `File_name` is indicated as starting after a line containing "`+(File_name`" in the line number field, and ending just before a matching "`+)File_name`" line. The included lines have "+" in the leftmost column of the line-number field, and those lines are numbered independently of the main source file.

An `Included` file inside an `Include` file has an extra "+" on each of its lines for each level of inclusion, except that line numbers take precedence over "+"s in the line-number field if and when the "+"s would otherwise intrude into the field.

The listing facility should be used in conjunction with the **`-Hnocpp`** option. Otherwise the output of the outboard C preprocessor will be listed; each `Include` file specified with the **`#include`** preprocessor statement is back substituted with no indication on the listing.

Example. Because a picture is worth a thousand words, a sample program listing appears on the next two pages, enhanced with boldface reserved words and followed by the optional assembly listing requested by **`-Hasm`** on the following compile command line:

```
hc queens.c  -Hlist  -Hasm  -Hnocpp
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                    Page 1
Copyright (C) 1983-87 MetaWare Incorporated.

Target:  4.3/RT                  (Code generator 2.7)
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+
             1 |/* From Wirth's Algorithms+Data Structures=Programs. */
             2 |/* This program is suitable for a code-generation    */
             3 |/* benchmark, especially given common sub-expressions*/
             4 |/* in array indexing.  See the Programmer's Guide for*/
             5 |/* how to get a machine code interlisting.           */
             6 |
             7 |pragma  Title("Eight Queens problem.");
             8 |
             9 |typedef enum {False,True} Boolean;
            10 |typedef int  Integer;
            11 |
            12 |#define Asub(I) A[(I)-1] /*C's restriction that array */
            13 |#define Bsub(I) B[(I)-2] /* indices start at zero     */
            14 |#define Csub(I) C[(I)+7] /* prompts definition of     */
            15 |#define Xsub(I) X[(I)-1] /* macros to do subscripting.*/
            16 |                              /* Pascal equivalents: */
            17 |static Boolean A[ 8]; /* A:array[ 1.. 8] of Boolean */
            18 |static Boolean B[15]; /* B:array[ 2..16] of Boolean */
            19 |static Boolean C[15]; /* C:array[-7.. 7] of Boolean */
            20 |static Integer X[ 8]; /* X:array[ 1.. 8] of Integer */
            21 |
            22 |void Try(Integer I, Boolean *Q) {
1 1         23 |   Integer J = 0;
1 1         24 |   do {
2 2         25 |      J++; *Q = False;
2 2         26 |      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
3 3         27 |         Xsub(I) = J;
3 3         28 |         Asub(J) = False;
3 3         29 |         Bsub(I+J) = False;
3 3         30 |         Csub(I-J) = False;
3 3         31 |         if (I < 8) {
4 4         32 |            Try(I+1,Q);
4 4         33 |            if (!*Q) {
5 5         34 |               Asub(J) = True;
5 5         35 |               Bsub(I+J) = True;
5 5         36 |               Csub(I-J) = True;
5 5         37 |               }
4 4         38 |            }
3 3         39 |         else *Q = True;
3 3         40 |         }
2 2         41 |      }
1 1         42 |   while (!(*Q || J==8));
1 1         43 |   }
            44 |pragma Page(1); /* Page eject requested. */
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                    Page 2

                Eight Queens problem.
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+
            45 |void main () {
1 1         46 |   Integer I; Boolean Q;
1 1         47 |   printf("%s\n","go");
1 1         48 |   for (I =  1; I <=  8; Asub(I++) = True);
1 1         49 |   for (I =  2; I <= 16; Bsub(I++) = True);
1 1         50 |   for (I = -7; I <=  7; Csub(I++) = True);
1 1         51 |   Try(1,&Q);
1 1         52 |pragma Skip(3); /* Skip 3 lines. */


1 1         53 |   if (Q)
1 1         54 |      for (I = 1; I <= 8;) {
2 2         55 |         printf("%4d",Xsub(I++));
2 2         56 |         }
1 1         57 |   printf("\n");
1 1         58 |   }
```

If the **-Hasm** option is specified, the source-annotated assembly listing on the next few pages is produced. (The page boundaries have been adjusted to fit the present page sizes.)

```
MetaWare High C Compiler 1.4                07-Jul-86 17:13:14 queens.c     Page 1
Copyright (C) 1983-86 MetaWare Incorporated.

Target: 4.2/RT   (Code generator 1.3)
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+
             1 |/* From Wirth's Algorithms+Data Structures = Programs.*/
             2 |/* This program is suitable for a code-generation     */
             3 |/* benchmark, especially given common sub-expressions */
             4 |/* in array indexing.  See the Programmer's Guide for */
             5 |/* how to get a machine code interlisting.
                                                          */
             6 |
             7 |pragma  Title("Eight Queens problem.");
             8 |
             9 |typedef enum {False,True} Boolean;
            10 |typedef int  Integer;
            11 |
            12 |#define Asub(I) A[(I)-1] /* C's restriction that array*/
            13 |#define Bsub(I) B[(I)-2] /* indices start at zero     */
            14 |#define Csub(I) C[(I)+7] /* prompts definition of     */
            15 |#define Xsub(I) X[(I)-1] /* macros to do subscripting.*/
            16 |                          /* Pascal equivalents:      */
            17 |static Boolean A[ 8]; /* A:array[ 1.. 8] of Boolean */
            18 |static Boolean B[15]; /* B:array[ 2..16] of Boolean */
            19 |static Boolean C[15]; /* C:array[-7.. 7] of Boolean */
            20 |static Integer X[ 8]; /* X:array[ 1.. 8] of Integer */
            21 |
            22 |void Try(Integer I, Boolean *Q) {
  1         23 |   Integer J = 0;
  1         24 |   do {
  2         25 |      J++; *Q = False;
  2         26 |      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
  3         27 |         Xsub(I) = J;
  3         28 |         Asub(J) = False;
  3         29 |         Bsub(I+J) = False;
  3         30 |         Csub(I-J) = False;
  3         31 |         if (I < 8) {
  4         32 |            Try(I+1,Q);
  4         33 |            if (!*Q) {
  5         34 |               Asub(J) = True;
  5         35 |               Bsub(I+J) = True;
  5         36 |               Csub(I-J) = True;
  5         37 |               }
  4         38 |            }
  3         39 |         else *Q = True;
  3         40 |         }
  2         41 |      }
  1         42 |   while (!(*Q || J==8));
  1         43 |   }
            44 |pragma Page(1); /* Page eject requested. */
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                    Page 2

                Eight Queens problem.
Levels  LINE #  |----+----1----+----2----+----3----+----4----+----5----+
             45 |void main () {
  1          46 |   Integer I; Boolean Q;
  1          47 |   printf("%s\n","go");
  1          48 |   for (I =  1; I <=  8; Asub(I++) = True);
  1          49 |   for (I =  2; I <= 16; Bsub(I++) = True);
  1          50 |   for (I = -7; I <=  7; Csub(I++) = True);
  1          51 |   Try(1,&Q);
  1          52 |pragma Skip(3); /* Skip 3 lines. */


  1          53 |   if (Q)
  1          54 |      for (I = 1; I <= 8;) {
  2          55 |         printf("%4d",Xsub(I++));
  2          56 |         }
  1          57 |   printf("\n");
  1          58 |   }
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                Page 3

                Eight Queens problem.
Addr  Object    Source Program and Assembly Listing
                        .globl  .oVncs
                        .set    .oVncs,0
                        .globl  _printf
                        .globl  _.printf
          #/* From Wirth's Algorithms+Data Structures = Programs */
          #/* This program is suitable for a code-generation     */
          #/* benchmark, especially given common sub-expressions */
          #/* in array indexing.  See the Programmer's Guide for */
          #/* how to get a machine code interlisting.
                                                              */
          #pragma  Title("Eight Queens problem.");
          #typedef enum {False,True} Boolean;
          #typedef int  Integer;
          ##define Asub(I) A[(I)-1]   /* C's restriction that array*/
          ##define Bsub(I) B[(I)-2]   /* indices start at zero
           */
          ##define Csub(I) C[(I)+7]   /* prompts definition of
           */
          ##define Xsub(I) X[(I)-1]   /* macros to do subscripting.*/
          #                           /* Pascal equivalents:
                   */
                  #static Boolean A[ 8];  /* A:array[ 1.. 8] of Boolean */
                        .data
0000              L00_DATA:
0000  00                .byte   0
                        .set    _A,L00_DATA+0
                  #static Boolean B[15];  /* B:array[ 2..16] of Boolean */
0001                    .space  7
0008  00                .byte   0
                        .set    _B,L00_DATA+8
                  #static Boolean C[15];  /* C:array[-7.. 7] of Boolean */
0009                    .space  15
0018  00                .byte   0
                        .set    _C,L00_DATA+24
                  #static Integer X[ 8];  /* X:array[ 1.. 8] of Integer */
0019                    .space  15
0028  00                .byte   0
                        .set    _X,L00_DATA+40
                  #void Try(Integer I, Boolean *Q) {
                        .text
0000                    .align  1
0000              L000:
                        .globl  _.Try
                  _.Try:
0000  D961 FFB4         stm     r6,-76(r1)
0004  6E00              mr      r14,r0
0006  6D10              mr      r13,r1
0008  C811 FFB4         cal     r1,-76(r1)
000C  6C20              mr      r12,r2
000E  6B30              mr      r11,r3
                  #   Integer J = 0;
0010  A4A0              lis     r10,0
                  #   do {
                  #      J++; *Q = False;
0012              L012:
0012  90A1              ais     r10,1
0014  A490              lis     r9,0
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                    Page 4

                Eight Queens problem.
Addr  Object    Source Program and Assembly Listing
0016  109B              stcs    r9,0(r11)
                #      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
0018  C82E FFA8         cal     r2,-88(r14)
001C  682A              cas     r8,r2,r10
001E  CE38 FFFF         lc      r3,-1(r8)
0022  B439              c       r3,r9
0024  0A2D              je      L07E
0026  63AC              cas     r3,r10,r12
0028  6723              cas     r7,r2,r3
002A  4637              lcs     r3,6(r7)
002C  B439              c       r3,r9
002E  0A28              je      L07E
0030  63C0              mr      r3,r12
0032  E23A              s       r3,r10
0034  6623              cas     r6,r2,r3
0036  CE36 001F         lc      r3,31(r6)
003A  B439              c       r3,r9
003C  0A21              je      L07E
                #        Xsub(I) = J;
003E  63C0              mr      r3,r12
0040  AA32              sli     r3,2
0042  E123              a       r2,r3
0044  39A2              sts     r10,36(r2)
                #        Asub(J) = False;
0046  DE98 FFFF         stc     r9,-1(r8)
                #        Bsub(I+J) = False;
004A  1697              stcs    r9,6(r7)
                #        Csub(I-J) = False;
004C  94C8              cis     r12,8
                #       if (I < 8) {
004E  89900016          bhex    L07A
0052  DE96 001F         stc     r9,31(r6)
                #           Try(I+1,Q);
0056  62C0              mr      r2,r12
0058  9021              ais     r2,1
005A  63B0              mr      r3,r11
005C  8DFFFFD2          balix   r15,_.Try       # Try
0060  C80E 0000         cal     r0,0(r14)
                #           if (!*Q) {
0064  402B              lcs     r2,0(r11)
0066  B429              c       r2,r9
0068  020B              jne     L07E
                #               Asub(J) = True;
006A  A491              lis     r9,1
006C  DE98 FFFF         stc     r9,-1(r8)
                #               Bsub(I+J) = True;
0070  1697              stcs    r9,6(r7)
                #               Csub(I-J) = True;
0072  89800006          bx      L07E
0076  DE96 001F         stc     r9,31(r6)
                #               }
                #           }
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                    Page 5

                Eight Queens problem.
Addr  Object    Source Program and Assembly Listing
                #         else *Q = True;
007A            L07A:
007A  A421              lis     r2,1
007C  102B              stcs    r2,0(r11)
                #         }
                #       }
                #   while (!(*Q || J==8));
007E            L07E:
007E  402B              lcs     r2,0(r11)
0080  9420              cis     r2,0
0082  0203              jne     L088
0084  94A8              cis     r10,8
0086  02C6              jne     L012
0088            L088:
0088  61D0              mr      r1,r13
008A  C961 FFB4         lm      r6,-76(r1)
008E  E88F              br      r15
0090  DF07DF68          .long   0xDF07DF68
                                        # First gpr=r6
0094  2D00              .short  0x2D00  # npars=2, off=0
                        .data   1
                        .globl  _Try
0058            _Try:
0058  00000000'         .long   L000
005C                    .align  2
                #   }
                #pragma Page(1); /* Page eject requested. */
                #void main () {
                        .text
0096                    .align  1
0096            L096:
                        .globl  _.main
                _.main:
0096  D9B1 FFC8         stm     r11,-56(r1)
009A  6E00              mr      r14,r0
009C  6D10              mr      r13,r1
009E  C811 FFC4         cal     r1,-60(r1)
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  queens.c                Page 6

                Eight Queens problem.
Addr  Object    Source Program and Assembly Listing
                #   Integer I; Boolean Q;
                #   printf("%s\n","go");
00A2  C82E FFEC         cal     r2,-20(r14)
00A6  C83E FFF0         cal     r3,-16(r14)
00AA  8DF00000'         balix   r15,_.printf
00AE  CD0E 0004         l       r0,4(r14)
                #   for (I =  1; I <=  8; Asub(I++) = True);
00B2  A4C1              lis     r12,1
00B4            L0B4:
00B4  94C8              cis     r12,8
00B6  0B09              jh      L0C8
00B8  A421              lis     r2,1
00BA  6BC0              mr      r11,r12
00BC  E1C2              a       r12,r2
00BE  63BE              cas     r3,r11,r14
00C0  898FFFFA          bx      L0B4
00C4  DE23 FFA3         stc     r2,-93(r3)
                #   for (I =  2; I <= 16; Bsub(I++) = True);
00C8            L0C8:
00C8  A4C2              lis     r12,2
00CA            L0CA:
00CA  D40C0010          ci      r12,16
00CE  0B09              jh      L0E0
00D0  A421              lis     r2,1
00D2  6BC0              mr      r11,r12
00D4  E1C2              a       r12,r2
00D6  63BE              cas     r3,r11,r14
00D8  898FFFF9          bx      L0CA
00DC  DE23 FFAA         stc     r2,-86(r3)
                #   for (I = -7; I <=  7; Csub(I++) = True);
00E0            L0E0:
00E0  C8C0 FFF9         cal     r12,-7(r0)
00E4            L0E4:
00E4  94C7              cis     r12,7
00E6  0B09              jh      L0F8
00E8  A421              lis     r2,1
00EA  6BC0              mr      r11,r12
00EC  E1C2              a       r12,r2
00EE  63BE              cas     r3,r11,r14
00F0  898FFFFA          bx      L0E4
00F4  DE23 FFC3         stc     r2,-61(r3)
                #   Try(1,&Q);
00F8            L0F8:
00F8  A421              lis     r2,1
00FA  C8BD FFC7         cal     r11,-57(r13)
00FE  63B0              mr      r3,r11
0100  8DFFFF80          balix   r15,_.Try       # Try
0104  CD0E 0008         l
                                r0,8(r14)
                #pragma Skip(3); /* Skip 3 lines. */
                #   if (Q)
0108  402B              lcs     r2,0(r11)
010A  9420              cis     r2,0
010C  0A11              je
```

```
MetaWare High C Compiler 1.4  07-Jul-86 17:13:14  cqueens.c                Page 7

                Eight Queens problem.
Addr  Object    Source Program and Assembly Listing
                                    L0012E
                #       for (I = 1; I <= 8;) {
010E  A4C1              lis     r12,1
0110            L00110:
0110  94C8              cis     r12,8
0112  0B0E              jh      L0012E
                #         printf("%4d",Xsub(I++));
0114  C82E FFF4         cal     r2,-12(r14)
0118  6BC0              mr      r11,r12
011A  90C1              ais     r12,1
011C  AAB2              sli     r11,2
011E  63BE              cas     r3,r11,r14
0120  CD33 FFC8         l       r3,-56(r3)
0124  8DF00000'         balix   r15,_.printf
0128  CD0E 0004         l       r0,4(r14)
012C  00F2              j       L00110
                #         }
                #   printf("\n");
012E            L0012E:
012E  C82E FFF8         cal     r2,-8(r14)
0132  8DF00000'         balix   r15,_.printf
0136  CD0E 0004         l       r0,4(r14)
013A  61D0              mr      r1,r13
013C  C9B1 FFC8         lm      r11,-56(r1)
0140  E88F              br      r15
0142  DF07DFB8          .long   0xDF07DFB8
                                        # First gpr=r11
0146  0D00              .short  0x0D00  # npars=0, off=0
                        .data   1
                        .globl  _main
005C            _main:
005C  00000096'         .long   L096
0060  00000000'         .long~
                                _printf
0064  00000058'         .long   _Try
0068                    .align  2
                        .data
0029                    .space  31
0048            .LITERALS.:
0048  25730A            .ascii  "%s\012"
004B  00                .byte   0
004C  676F              .ascii  "go"
004E  00                .byte   0
004F                    .space  1
0050  253464            .ascii  "%4d"
0053  00                .byte   0
0054  0A                .ascii  "\012"
0055  00                .byte   0
                        .data
0056                    .space  2
No user errors   4 unprinted warnings
End of processing, 07-Jul-86 17:13:19                queens.c
```

11. MAKING CROSS REFERENCES

This chapter explains how to use the **hcxref** command to generate a cross-reference listing of one or more High C modules.

11.1 Features of the Cross Reference

Cross references have the following features:

References to source files. All cross-reference information refers to line numbers within files compiled, as opposed to line numbers within a listing. Therefore no listing is necessary to use the cross reference.

Include files. Included source files are handled properly. That is, they do not interfere with the process, and their names are included correctly in the results.

Assignments versus uses. References that assign values into variables are distinguished from references that use values of variables.

Annotated listing. It is possible to generate an annotated source listing of one or more program files. The listing contains cross-reference information to the right of the source text listed.

Multi-module cross references. A cross reference can span multiple compilation units by cross-referencing many modules at once and showing references from one module into the other. Thus, a single cross reference can be produced for a program that is broken up into separately compiled modules.

Inter-module usage summaries. A list of the names that one module uses that are located in other files can be produced, organized by file. This helps one understand the module interconnectivity of a large program.

11.2 Using the hcxref Command

The **hcxref** command processes one or more High C source files and produces a cross-reference listing on standard output. The listing consists of up to four components as described in §11.3 *Cross-Reference Format* below.

The command has the following form:

hcxref [**-ilmpus**] [*preprocessor_options*]... *files*...

where *files* denotes one or more High C source files, and *preprocessor_options* denotes zero or more preprocessor options (for example, `-Idir` or `-Dname`) that are required when compiling the files.

The **-l** option causes a listing of the source files to be generated, annotated with cross-reference information. `Include` files are not expanded in the listing unless **-i** is also specified.

The **-m** option causes a listing to be produced, for each module M, of the names referenced in M that were defined elsewhere.

Names that are declared but not referenced do not appear in the cross reference unless the **-u** option is specified.

The **-p** option causes the outboard C preprocessor to be invoked on each source file. The output of the preprocessor is then processed by the cross referencer instead of the source files themselves. If this option is not specified, the inboard preprocessor is used. This option is analogous to the **-Hcpp** option of the **hc** command. The **-l** and **-i** options are ignored when used in conjunction with **-p**.

The **-s** option specifies that various statistics relating to the cross reference are to be printed.

The **hcxref** command invokes the High C compiler in a special mode to generate the cross-reference information. Therefore, if any of the source files contains errors, appropriate diagnostics are generated.

11.3 Cross-Reference Format

Components. Each cross reference is self-documenting and consists of four components:

(1) An alphabetized list of all names declared in the program, together with an ordered list of all the references to each name.

(2) An alphabetized table of all files used in the program and a file reference number for each.

(3) A list for each module `M` of all the names used by `M` that are declared in other files — if requested.

(4) An annotated cross reference for each module — if requested.

When the components are produced:

Item (1) is always produced.

Item (2) is produced if the cross reference involves more than one file; this happens if more than one module is cross-referenced, or if any compiled include files were involved in the modules being cross-referenced.

Item (3) is produced if the **-m** option is specified.

Item (4) is produced if the **-l** option is specified.

What each component consists of:

Item (1) presents the following information for each distinct name in the program:

- The line and column number of the declaration of the name. If the name occurs in a compiled `Include` file, or if several modules are being cross-referenced, the file number is also given.
- The declared name `N`, and its *owner*: the name of the function that contains `N`'s declaration.
- Information about the named object, such as its storage class (**`static`**, **`extern`**, **`typedef`**, **`register`**, etc.) and in some cases, the object's type.
- The numbers of any lines containing references to the name. If the references are not in the module being cross-referenced, (they may be in an `Included` file), or if several modules are being cross-referenced, the line numbers are presented in the format `fn<I>` where `n` is the number of the file containing the references and `I` is the list of line numbers. Occasionally the entry in this field is of the form `resolved at` *`ref`* where *`ref`* is a line number or `fn<...>` reference as just described. This means that the name was introduced by an **`extern`** declaration whose actual definition was given at *`ref`*.
- References that assign, or may assign, a value to a variable are marked with the character `*`.

Item (2) presents the correspondence between file numbers and file names. References in items (1) and (3) use the file number rather than file names, to keep the listing brief. Item (2) is used to determine the corresponding file name.

Item (3) is optional. It is requested by the **-m** option. The output produced is a listing for each module `M` of the names used by `M` that are declared in other files. The list is organized by file. This is useful for determining the interconnectivity between modules. For example, if module `M1` refers to no function names within module `M2`, it may be possible to overlay the code of `M1` and `M2`.

In Items (1) and (3) a reference to a name `N` declared at reference point `P` is changed to a reference to a point `P'`, if the definition at `P'` resolves the declaration at `P`. Typically this happens when `N` is declared in an interface file `F`, is used in a module `M`, and is defined at `P'` in a module `M'`. The module usage in Item (3) shows that `M` refers to `P'` in module `M'`, *not* `P` in interface file `F`. That is, one gets references to the implementations rather than the interfaces through which they were supplied.

Item (4) is optional and is is generated by the **-l** option. The result is a line-numbered listing of the source of the compiled program, with each line annotated on the right with the line numbers of the definitions of names used on the line.

If *n* names are used on the line, *n* line numbers appear to the right of the line, corresponding positionally. A line number alone is a reference into the file being listed. If the letter `i` appears instead, the name referenced is an intrinsic, such as `_find_char` or `_abs`. Finally, a line number followed by `f` and another number means

that the name was declared in a file other than the one being listed; the file number can be used to discover that file's name in Item(2). `Line#fFile#` was used instead of `File#<Line#>` as in Item (1) for brevity.

11.4 Distinction of File Names

In a multi-module cross reference, a particular interface file may have been included by several modules because each of the modules being cross referenced needs the resources in that file. The cross referencer assumes that a repeated declaration of a name in a compiled `Include` file is the same declaration if it appears at the same line and column number of the same `Include` file.

For purposes of determining "sameness of `Include` files" the cross referencer uses the text of the file name including the path. Therefore, to cross-reference several modules successfully, do not use different names for the same `Include` file.

For example, if module `M1` includes `../utils/trees.h`, and `M2` includes `/prog/utils/trees.h`, and if these two references denote the same file, the cross referencer will *not* recognize them as the same.

12. DIAGNOSTIC MESSAGES

Messages from the High C compiler report (a) file I/O errors, (b) system errors, and (c) user errors and warnings.

12.1 File I/O Errors

File I/O errors are fatal.[8] They can occur in attempting to open a non-existent file or in writing a compiler output file when not enough file space is available. The errors likely to be seen are:

`Unable to open file fff: file not found.`

This message is produced when any input source file, such as that specified on the command line or in an `Include` pragma, cannot be found.

This message is produced twice: it is written once to standard output and once to standard error. If standard output is not redirected, the message appears on the screen twice.

`***Error occurred on writing instruction file: ...`
`***Error occurred on writing object file: write failed.`

Usually caused by too little space on disk. Remove unnecessary disk files and try again.

12.2 System Errors

System errors are fatal[8] and should rarely occur. They take the form:

`>>>> S Y S T E M E R R O R n <<<<, in Module:Function`
Error message text.

where **n** numbers the occurrences of system errors, ***Module*** is the module name, and ***Function*** is the function name. The only system error messages with which the user should be concerned are:

`Dynamic array allocation/reallocation failed.`
`Out of memory.`

This error indicates that the user's virtual memory quota was exceeded.

`Recover: Exceeded the following limit: Limit.`

In repairing a syntax error, a table overflowed. The table limit is fixed, so no increase in memory can improve the situation. Repair the error.

There are many other system error messages that the compiler could produce, but they are associated with internal compiler errors or inconsistencies that should not occur.

Stack dump. Compiler system errors are always accompanied by a call-stack dump. The dump can usually be ignored, but when reporting a problem to the support staff, the history of called functions can be helpful; include a listing of the dump in any written correspondence. The following is a sample dump:

```
>>>>> S Y S T E M  E R R O R 1 <<<<<, in Scanner:Read_scan_tables
No scan tables found.
                                  Line
Routine                 File      /Off Addr  Parms...
syserr                  syserr.p   66   54d3a c098,c080,0,66290,66320
read_scan_tables        stread.p   69   bef2  2004a4c,fffa60,44ed,663c4
get_scan_tables.stread  stread.p   39   c005  663c4,66290,fffadc,14e8,1
analdrvr                analdrvr.p 19   44ed  1,663c4,66416,0,1,186dc
initialize_prefix.sk    skelinit.p 2dc  14e8  fffaec,115a,fffaf4,59645
doit                    skeldrvr.p e    111f  fffaf4,59645,fffb04,4d,3
pp_main                 skeldrvr.p 6    115a  fffb04,4d,3,fffb08,fffb18
_main                   ppinit     1d   59645 3,fffb08,fffb18,ufffb38
start                              3d   4d    fffb3d,fffb45,0,fffb4b
Error was severe.  Program terminated.
```

---

8. Fatal errors may result in compiler temporary files being left in the `/tmp` directory. They should be removed.

The **Routine** and **File** columns are usually sufficient alone when reporting a problem to support personnel.

System errors due to a bug in the compiler's code generator are accompanied by a line "`Code was being generated for program text near Ln/Cm.`" following the call-stack dump. This helps isolate the program text causing the problem and may facilitate reducing the problem program to a few lines, which then can be easily sent to compiler support personnel.

NOTE: ***Code generator errors can frequently be "cured" by inserting a label before the line causing the problem.*** Even if this cures the problem, please report the problem to support personnel.

12.3 User Errors and Warnings

User error messages are grouped in the three categories (1) lexical, (2) syntactic, and (3) constraint. Warnings do not suppress object file generation; errors always do. Also, some diagnostics that are warnings become errors when the compiler is run in ANSI mode.

Messages that report errors terminate compilation after the phase issuing the diagnostic, so errors that would otherwise have been detected by later phases are not reported until all earlier errors are repaired and the compiler is reinvoked.

All user diagnostics are accompanied by the file name, a line number **n**, and column number **m**, in the form "`"filename", Ln/Cm`", reporting where the error was detected. In addition, when **-Hlist** is specified on the command line, as assumed in the examples below, lexical and syntactic errors are generally accompanied by the erroneous line with a caret "^" beneath it at the point of error detection. Error messages begin with "`E`" and warnings with "`w`", and usually occupy a single line.

Lexical error messages are produced when an improperly formed word is detected, such as a string with a missing closing quote.
Example:

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5
               1 |void main() {
1 1            2 |   char *S;
1 1            3 |   S = "Hello;
             C15 ---------------^
E "file", L3/C15:  (lexical)  Unexpected end-of-line encountered.
1 1            4 |   }
```

Syntactic error messages are produced for programs that are ill-formed on the phrase level, such as a missing "`;`" or inserted spurious symbol. The message is accompanied by a statement of the `REPAIR` that the compiler effected so it could keep processing input.
Example:

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5
               1 |void main() {
1 1            2 |   printf "Hello");
              C11 ----------^
1 1            3 |   }
E "file", L2/C11:  (syntactic)  unexpected symbol:'<STRING>':"Hello"
REPAIR:   '(' was inserted before '<STRING>':"Hello"@L2/C11
```

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are nearly 200 such diagnostics, each of which is meant to be self-explanatory. Most of them prevent the generation of object code, but some are merely warnings and are intended to assist the programmer.
Examples:

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5
               1 |void main() {
1 1            2 |   int i;
1 1            3 |   i = Undeclared_identifier;
1 1            4 |   }
E "file", L3/C8:   Undeclared_identifier: This is undeclared.
1 user error    No warnings    453K of memory unused.
```

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5
             1 |void main() {
1 1          2 |   int i, Unused;
1 1          3 |   i /= 0;
1 1          4 |   }
w "file", L2/C8:   i: Variable is set but is never referenced.
w "file", L2/C11:  Unused: Variable is never used.
E "file", L3/C6:   Division by zero.
1 user error   2 warnings     457K of memory unused.
```

12.4 Error and Warning Messages

This section presents all compiler diagnostic messages, except automatically generated lexical and syntactic messages, in alphabetical order, with explanations where appropriate.

`(lexical) Unexpected ...`

`(syntactic) Unexpected symbol: ...`

`"=" used where "==" may have been intended.`
"=" was detected as an operator in a Boolean expression, such as "`if (x = y) (...)`". Often this is a mistake, as "`if (x == y) (...)`" was intended.

`"auto" must appear within a function.`
Storage class **auto** cannot be given for declarations that do not appear within a function.

`"break" must appear within while, do, for, or switch.`

`"case" must appear within a "switch".`

`"continue" must appear within while, do, or for.`

`"default" must appear within a "switch".`

`"pragma Data" active at end of module.`

`"pragma Data" active at end of function.`
A `pragma Data(...);` was given in a module or function, with no terminating `pragma Data;`. This is permitted but the programmer may have forgotten to supply the terminating pragma, thus perhaps including more data declarations in a data segment than intended.

`"register" is the only allowable storage class for a parameter. Ignored.`
In a function definition or declaration, a storage class other than **register** was given, such as in `int f(i) static i; {...}`.

`"register" must appear within a function.`
Storage class **register** cannot be given for declarations that do not appear within a function definition.

`"void" is illegal here.`

`A bit field is not valid as an argument to &.`
One cannot take the address of a bit field, since such a field is not necessarily on a byte boundary.

`A bit field is not valid as an argument to sizeof.`
Since bit fields need not occupy an integral number of bytes, taking their **sizeof** is prohibited.

`A function may not return a function (but may return a pointer thereto).`

`A function may not return an array (but may return a pointer thereto).`

`A function may not return an incomplete type.`
A function cannot return a **struct** or **union** type whose fields have not yet been specified. For example, `struct s; struct s *f() {...}` is legal since `f` returns a *pointer* to an incomplete **struct** type, but `struct s; struct s g() {...}` is illegal.

`A functionality typedef cannot be used in a function definition.`
`typedef int f(); f g {return 3;}` is illegal: the type definition for `f` cannot be used to specify that `g` is a function.

`A parameter may not be a function (but may be a pointer thereto).`

`A parameter name must be given here.`
For function definitions, parameter names must be supplied. Thus, for example, **void** `f(`**int**`,` **float** `g) {...}` is illegal because the first parameter lacks a name.

`A register-class function makes no sense.`
For example, **register** `f() {...}` is illegal.

`An array may not contain functions (but may contain pointers thereto).`

`An array must have a positive number of elements.`

`An array of objects of an incomplete type is illegal.`
An array cannot contain a **struct** or **union** type whose fields have not yet been specified. For example, **struct** `s;` **struct** `s *a[10];` is legal since "a" contains *pointers* to an incomplete **struct** type, but **struct** `s;` **struct** `s b[10];` is illegal.

`An object of type` ttt `cannot be initialized.`

`Argument to "`**#include**`" must be a string.`

`Argument type` ttt `is not compatible with formal parameter type` ttt'.
An attempt was made to pass an argument of a wrong type to a function, such as passing a **float** for a parameter that is a **struct**. When using standard C function definitions, this is a warning only, since C permits such mismatches, but when using prototype syntax, it is an error. This warning provides the security of Pascal function call semantics.

`Array size exceeds addressability limits.`

`Bit fields must fit in a register or register pair.`

`Cannot dereference a pointer to` **void**.
Type ***void** was introduced as a means of defining a "generic pointer" compatible with other pointers. But there is no such thing as an object of type **void**. Therefore, dereferencing a pointer to **void** is illegal.

`Cannot initialize a` **typedef**.
Something like **typedef int** `T = 1;` was attempted.

`Cannot initialize an imported variable.`
Something like **extern int** `T = 1;` was attempted. A variable may be initialized only by its definition.

`Cannot take` **sizeof** `a function type.`

`Cannot take` **sizeof** `an incomplete type.`
The **sizeof** a **struct** or **union** type whose fields have not yet been specified is not known. For example, **struct** `s; (...)` **sizeof**`(`**struct** `s) (...)` is illegal since the size of the structure is unknown.

`Cannot take` **sizeof** `type` **void**.
There are no objects of type **void**, therefore taking **sizeof void** makes no sense.

`Cannot take the address of a` **register** `variable.`

`Declared type is never referenced.`

`Divide by zero.`
This was detected in a constant expression at compile time.

`Enclosing function's return type is "`**void**`"; therefore nothing may be returned.`
return `E;` for some expression `E` was found in a function whose return type is **void**.

`End of file encountered within` **#if** `construct.`

`End of file encountered within arguments to a macro. Probably a missing right parenthesis.`

`End of file encountered within macro definition.`

`End of file encountered within macro formal parameter list.`

`Expression has no side effect and has been deleted.`
An expression used in a statement context has no side effect; therefore the expression is useless. For example, `2+3;`.

`External function is never referenced.`

`Fewer arguments given than function has parameters.`

`for` `loop will never execute.`

`Function called but not defined.`

Any function that was called but not defined is noted as a warning. Although such practice is permissible in C, especially useful when calling library functions, a common error is to misspell a function name. The error goes undetected until link-time without this warning. Furthermore, errors in parameter linkage can occur when a call is made to an undefined function. We recommend that the library "`.h`" header files always be included to get parameter checking, and that function prototypes be used for external function declarations, rather than making use of the "feature" of C for calling undefined functions.

`Function expected.`

The expression `f` preceding the arguments in a function call `f(...)` must denote a function.

`Function parameter names are allowed only on function definitions, not declarations.`

`int` `f(a,b,c);` is a function declaration that names the parameters `(a,b,c)`. This is illegal unless function prototype syntax is used, as in **`int`** `f(`**`int`** `a,` **`int`** `b,` **`int`** `c);`.

`Function return value never specified within function.`

A function with a non-**`void`** return type contains no **`return`** statement. This typically happens with "old" C programs that did not use **`void`** to indicate that a function returns nothing.

`Functions may not be nested.`

In ANSI-Standard C, functions cannot be declared within functions. In High C they can. This message is produced when the compiler is doing ANSI checking.

`Identifier required after` **`#ifdef`** `or` **`#ifndef`**`.`

`Identifier required. Pragma ignored.`

`Incompatible tag reference: The` ttt `tag class does not match the tag class` ttt' `defined at Ln/Cm.`

Something like **`struct`** `s;` **`union`** `s {`**`int`** `x;};` was encountered. The tag `s` cannot simultaneously be the tag for a **`struct`**, **`union`**, and/or **`enum`**.

`Incomplete type: the` **`struct/union`** `type at Ln/Cm must be completed before it can be used here.`

A reference has been detected to a field of a **`struct`** or **`union`** type whose fields have not yet been specified.

`Incorrect number of parameters to macro. Macro invocation ignored.`

The number of arguments to a macro must agree exactly with the number of parameters in its **`#define`**.

`Integer constant exceeds largest` **`unsigned`** `number.`

`Invalid digit in non-decimal number: X.`

`Local function is never referenced; no code will be generated for it.`

A function of storage class **`static`** is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function and it is essentially deleted.

`Lower bound of range is greater than upper bound.`

This can only happen in High C **`case`** statements where range expressions are allowed as labels (an extension). Macro name must be an identifier.

`Macro parameter must be an identifier.`

`Members cannot be of an incomplete type.`

A **`struct`** or **`union`** cannot contain a **`struct`** or **`union`** type whose fields have not yet been specified. For example, **`struct`** `s;` **`struct`** `t {`**`struct`** `s *p;}` is legal since `p` is a *pointer* to an incomplete **`struct`** type, but **`struct`** `s;` **`struct`** `t {`**`struct`** `s p;}` is illegal.

`Mismatched` **`#if-#elif-#else-#endif`**`.`

`More arguments given than function has parameters.`

`Must be a compile- or load-time computable expression.`
The initializers for a static variable must be determinable when a program is loaded.

`Must be a compile-time computable constant.`

`Must be a pointer.`

`Must be a scalar (int, char, floating, or pointer) type.`

`Must be a static variable reference.`

`Must be a string.`

`Must be a struct or union.`

`Must be a type.`

`Must be an identifier.`

`Must be an integral int or char type.`

`Must be of a pointer type.`

`Must be of an extended-function type.`

`Named parameter association is prohibited for this function since its declaration near Ln/Cm does not name all parameters.`
An attempt was made to call a function `F` using named parameter association, but `F`'s declaration did not name all of its parameters. Example:

```
void F(int a,float);  … F(a=>37,   3.3);
   /*Illegal.*/
void F(int a,float b);… F(a=>37,b=>3.3);
   /*Fine.*/
```

`No "pragma Data" is active.`
`pragma Data;` was encountered without a preceding, and matching, `pragma Data(…);`.

`No member is declared here.`
A declaration with no declared object was found within a `struct` or `union`. For example,

```
struct s {int; float; struct t {int y};}
```

contains three declarations, none of which declare an object. However, this construct is not entirely vacuous because the declaration of `struct t` is visible outside of `struct s` and therefore can be used to declare objects of type `struct t`.

`No object may be of type void.`

`No parameter declarations may be given here.`
In defining a function using prototype syntax, where the parameter types were specified in the parameter list, an attempt was made to re-declare the parameters following the parameter list. For example, `int x,y;` is illegal in `void f(int x, int y) int x,y; { … }`.

`Non-decimal constant exceeds largest unsigned number.`

`Only a parameter may be declared here.`
Preceding a function definition's `{`, only the function's parameters may be declared.

`Only fields of type "unsigned int" or "unsigned long int" are supported.`
Bit fields may be of only these two types. Any bit field of another type is coerced to one of them, depending upon the size of the bit field.

`Only one "default" is permitted in a "switch".`

`Operand type inappropriate for operator.`
An inappropriate operand was detected for a built-in operator such as `&, |, ~`, etc. For example, `float f1,f2; (…)f1 = f1 & f2;` is illegal: `&` requires integral operands.

`Parameter not found or specified more than once.`
In a function call using named parameter association, a parameter was named twice, or a non-existent parameter was referenced.

`Parameter ppp not supplied.`
In a function call using named parameter association, parameter `ppp` was not given an argument value.

`Parameter separator must be a comma.`

In a **#define** of a macro with parameters, parameter names must be separated by commas. For example, **#define** `M(a b) c` is illegal; `a,b` is required.

`Pointer dereferencing disallowed in static context.`

"pragma `Code" may not occur within a function.`

The `Code` pragma must appear only at the outermost declaration level — outside of all functions.

`Pragma has too few parameters.`

`Pragma has too many parameters.`

`Previous "pragma Data" is still active.`

pragma `Data(...);` was given in the context of an already active **pragma** `Data(...);`. Insert **pragma** `Data();` preceding the offending pragma to "turn off" the active pragma.

`Real constant has too many digits.`

`Result of comparison never varies.`

An expression was found whose operands, while they are not all constants, are such that the value of the expression is always the same. For example, an expression of type **unsigned int** is never less than zero.

`Right operand of shift operator is negative.`

`Since the first parameter was specified by the type "void", there may be no other parameters.`

The special syntax exemplified by **int** `f(void);` denotes a function `f` taking no parameters. Because of this, no parameter can be specified after **void**: **int** `f(void, float, int);` is illegal.

`Size change in cast involving pointer type: casted-to type` ttt `is not the same size as casted-from type` ttt'.

`Specified storage class for this declaration is unnecessary and was ignored.`

In a declaration such as **static struct** `s{int x;};`, the storage class **static** is useless since no object was declared.

`Static initialization of bit fields is not supported.`

`Storage-class nonsensical for function definition.`

`String too long for initialized array.`

`Structure has no contents (is of size zero).`

`Subscripted expression must be an array or pointer.`

`The 2nd and 3rd operands of a conditional expression must be both arithmetic, or of the same type, or one a pointer and the other zero.`

`The declarator must be a function. This declaration has been discarded.`

A declaration such as **int** `f {...};` was encountered, where a function body `{...}` was given for a non-function.

`The rest of this line is extraneous.`

`The sign (`**signed/unsigned**`) has been specified more than once.`

`The storage-class (`**auto, extern,** `etc.) has been specified more than once.`

`The width (`**long/short**`) has been specified more than once.`

`This "return" should return a value of type` ttt `since the enclosing function returns this type.`

`This can be of an incomplete type only if it is "extern" or has an initializer supplying its size.`

`This code will never be executed.`

`This construct would have been deleted as an optimization had it contained no labels.`

A construct such as **while** `(0) {...}` was detected but cannot be deleted due to the presence of one or more labels within `{...}`. This is questionable programming practice at best.

`This function declaration is inconsistent with the "int"-returning function declaration imputed at Ln/Cm.`

A function called before it is declared is assumed to be a function returning **int**, and any subsequent declaration of the function must declare it to be so. For example, `main () { (...) f(3);(...) }` **void** `f() {...}` is illegal since `f` was called before being defined and therefore assumed to return **int**.

`This function declaration is inconsistent with the declaration at Ln/Cm.`

`This is already defined as a macro. Redefinition ignored.`

A redefinition of a macro is permitted only if the redefinition agrees exactly with the previous definition. To otherwise redefine a macro, use **#undef** to explicitly undefine the macro before re-defining it.

`This is multiply declared.`

`This is permissible only in conjunction with "int" or "char".`

`This is permissible only in conjunction with "int" or "double".`

`This is permissible only in conjunction with "int".`

`This is undeclared.`

`This may not be a pointer to a function (but may be a pointer to an object).`

`This tag name is more than 80 characters long.`

`This type lacks a tag and hence cannot be used.`

A declaration such as **struct** `{int x;};` was encountered. Without a tag the **struct** cannot be referenced and hence is useless.

`Toggle name required. Pragma ignored.`

`Too many initializers here.`

`Type` ttt `is not assignment compatible with type` ttt`'.`

(a) In an assignment expression, the right operand of type `ttt` may not be assigned to the left operand of type `ttt'`.

(b) In a function call, an argument of the type `ttt` may not be passed to a function that expects a parameter of type `ttt'`.

`Type` ttt `is not compatible with type` ttt'.

In a comparison, the left operand of type `ttt` may not be compared with the right operand, of type `ttt'`.

`Unexpected symbol in expression. Line ignored.`

`Unknown preprocessing directive.`

`Unrecognizable Data class. Static assumed.`

`Unrecognizable field name.`

`Unrecognizable pragma name. Pragma ignored.`

`Unrecognizable toggle name. Pragma ignored.`

`Up-level reference to a` **register**`-class variable is not allowed.`

`Variable is never used.`

`Variable is referenced but is never set.`

`Variable is set but is never referenced.`

`Variable is referenced before it is set.`

`Variable required.`

In this context a so-called "lvalue" is required but was not found. An *lvalue* is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to `&`, `++`, and `--`. The rules of C require the automatic conversion of some objects into non-lvalues. For example, the operand of `&` must be an lvalue, so **int** `i = &(a+b)` produces the "`Variable required.`" diagnostic.

A common cause of this message is the use of a construct such as:

```
int * p;
c = *((char*)p)++;
```

which is legal on most PCC compilers, but disallowed by the Standard. Use instead:

```
int * p;
c = *(*(char**)&p)++;
```

to circumvent the restriction.

`Zero-length bit fields may not be named.`

A declaration such as **struct** `{int i:0,j:2};` was encountered. "`i`" must be omitted. As is, it is possible to refer to the field. Such a reference would be illegal.

`{...} inappropriate here for initializing a scalar.`

Appendix A

CROSS-JUMPING OPTIMIZATIONS

MetaWare compilers support an optimization that usually obtains a 2% to 5% reduction in code size and is often accompanied by a decrease in execution time. The optimization is known as "cross-jumping". It and the two toggles that control it are explained here.

Consider the following source code:

```
        if (!eof) readbytes(&buf,&cnt,512);  /*Code C.*/
/*L:*/  while (cnt > 0) {
            writebytes(&buf,cnt);
            if (!eof) readbytes(&buf,&cnt,512); /*Code C'.*/
            } /*Implicit jump back to the implicit label.*/
```

The compiler can improve the code size of this program without any loss in execution speed by effectively rewriting the code as:

```
Top:    if (!eof) readbytes(&buf,&cnt,512); /*Code C = C'.*/
/*L:*/ if (cnt > 0) {
           writebytes(&buf,cnt);
           goto Top;
           }
```

The optimization involves the recognition of some code `C` immediately preceding a jump `j` to some label `L`, where some code `C'` identical to `C` immediately precedes `L`. The transformation consists in deleting `C` and replacing `j` with a jump to `C'` instead:

Original Code		*Transformed Code*	
some code C		jmp L'	
jmp L			
...		...	
some code C'	L':	some code C = C'	
L: ...	L:	...	

This optimization is called "cross jumping" or "tail merging" in the compiler literature, since it was first invented to handle common code at the ends of the arms of conditional statements, and was effected by jumping across from one arm to the other, that is, by merging the tails of the two arms. It is surprisingly effective and always saves code space while never giving up execution speed.

Here we include another optimization under that name as well. The second optimization is even more effective, but gains its (sometimes considerable) code space in trade for a small loss of speed. Consider the program fragment:

```
if (buf[cnt]==0) g(&buf);
   else if (buf[cnt]== '\n') {buf[cnt] = 0; g(&buf);}
   else ...
```

The compiler effectively transforms this into:

```
if (buf[cnt]==0) goto L';
   else if (buf[cnt]=='\n') {buf[cnt] = 0; L': g(&buf);}
   else ...
```

Here, both occurrences of `g(&buf);` precede a jump to the statement following the entire conditional. One of the instances of `g(&buf);` is replaced with a jump to the other, saving the code space for the call to `g` at the expense of inserting an additional jump. Opportunities for this kind of optimization are even more frequent than the standard cross-jumping optimization. In general, the optimization can be depicted as follows:

Original Code		*Transformed Code*	
	some code C		**jmp** L'
	jmp L		
	...		...
	some code C' (= C)	L':	some code C = C'
	jmp L		**jmp** L
	...		...
L:	...	L:	...

Both optimizations are turned `On` by default. Both may be disabled by turning `Off` the toggle `Optimize_xjmp`, with either **-Hoff**=`Optimize_xjmp` on the compiler execution line, or including **pragma** `Off(Optimize_xjmp);` in the program. The second of the two optimizations can be disabled by turning `Off` the toggle `Optimize_xjmp_space`, so named because the second optimization saves space but always increases execution time.

During the development phase of a project, it may be desirable to turn `Optimize_xjmp Off`. The reason is that the optimization can cause such a contortion of code that using debuggers, whether assembly-language level or line-oriented symbolic, can be difficult. As a case in point, consider the following program, which compares the fields of two different structures to determine if they are the same:

```
union {
   struct {int x,y;}              f1;
   struct {int a,b,c;}            f2;
   struct {int e,f;}              f3;
   struct {int g,h; int i[10];}   f4;
   } u1,u2;

int f(i) int i; {
   switch(i) { /* What kind of structure to compare? */
      case 1: return   u1.f1.x == u2.f1.x &&
                       u1.f1.y == u2.f1.y;
      case 2: return   u1.f2.c == u2.f2.c &&
                       u1.f2.a == u2.f2.a &&
                       u1.f2.b == u2.f2.b;
      case 3: return   u1.f3.e == u2.f3.e &&
                       u1.f3.f == u2.f3.f;
      case 4: return   u1.f4.g == u2.f4.g &&
                       memcmp(u1.f4.i,u2.f4.i,
                        sizeof(u1.f4.i)) !=0;
      case 5: return   u1.f4.h == u2.f4.h &&
                       memcmp(u1.f4.i,u2.f4.i,
                        sizeof(u1.f4.i)) !=0;
   } }
```

Here cases 1 and 3 are recognized as being identical, and matching the tail end of case 2. Furthermore, cases 4 and 5 share a common tail. Compiling the code produces the tightly-coded result presented next. Even a skilled assembly-language programmer would rarely have the patience to produce such highly optimized code:

```
#int f(i) int i; {
#    switch(i) { /* What kind of structure to compare? */
        .text
        .align  1
L000:
        .globl  _.f
_.f:
        stm     r12,-52(r1)
        mr      r14,r0
        mr      r13,r1
        cal     r1,-52(r1)
        mr      r12,r2
        mr      r15,r12
        sis     r15,1
        cli     r15,4
        jh      L0C0
        a       r15,r15
        get     r2,$L02A
        a       r15,r2
        lhas    r15,0(r15)
        a       r15,r2
        br      r15
L02A:
        .short  L052-L02A
        .short  L034-L02A
        .short  L052-L02A
        .short  L074-L02A
        .short  L08C-L02A
#       case 1: return u1.f1.x == u2.f1.x &&
#                      u1.f1.y == u2.f1.y;
#       case 2: return u1.f2.c == u2.f2.c &&
L034:
        get     r2,$_u1
        ls      r3,8(r2)
        get     r4,$_u2
        ls      r5,8(r4)
        c       r3,r5
        jne     L0BE
        bx      L064
        ls      r3,0(r2)
#                      u1.f2.a == u2.f2.a &&
#                      u1.f2.b == u2.f2.b;
#       case 3: return u1.f3.e == u2.f3.e &&
L052:
        get     r2,$_u1
        ls      r3,0(r2)
        get     r4,$_u2
L064:
        ls      r5,0(r4)
        c       r3,r5
        jne     L0BE
        ls      r2,4(r2)
        ls      r3,4(r4)
        c       r2,r3
        jne     L0BE
        j       L0BA
#                      u1.f3.f == u2.f3.f;
#       case 4: return u1.f4.g == u2.f4.g &&
L074:
        get     r2,$_u1
        ls      r3,0(r2)
        get     r4,$_u2
        bx      L0A0
        ls      r5,0(r4)
#       memcmp(u1.f4.i,u2.f4.i,sizeof(u1.f4.i)) !=0;
#       case 5: return u1.f4.h == u2.f4.h &&
L08C:
        get     r2,$_u1
        ls      r3,4(r2)
        get     r4,$_u2
        ls      r5,4(r4)
L0A0:
        c       r3,r5
        jne     L0BE
        inc     r2,8
        cal     r3,8(r4)
        cal     r4,40(r0)
```

```
        balix   r15,_.memcmp
        l       r0,4(r14)
        cis     r2,0
        je      L0BE
L0BA:
        lis     r2,1
        j       L0C0
L0BE:
        lis     r2,0
L0C0:
        mr      r1,r13
        lm      r12,-52(r1)
        br      r15
        .long   0xDF07DFC8    # First gpr=r12
        .short  0x1D00        # npars=1, off=0
        .data   1
        .globl  _f
_f:
        .long   L000
        .long   _memcmp
        .align  2
        .data
```

In summary,

1. Cross-jumping is an amazingly effective optimization.
2. Toggle `Optimize_xjmp` is set `On` by default, and turning it `Off` disables all cross-jumping.
3. Toggle `Optimize_xjmp_space` is `On` by default, and turning it `Off` disables cross-jumping optimization that decreases space at the expense of time.

The cross-jumping optimization adds perhaps 20% to the execution time of the code generator phase of the compiler, thus perhaps 3% overall.

Index

Starting below is a "permuted key word in context" index for this document. In the center column is the particular key word W being indexed, in the context of a phrase or sentence containing W. The phrase appears to the left and right of W.

Occasionally the text of the phrase preceding W does not fit in the space to the left of W. In that case the index entry looks like

is text that was too long to precede the WORD being indexed. This 7

where the first word "This" of the sentence did not fit on the left. Similarly the text to the right of W can be crowded:

right. This WORD is followed by too much text on the 7

where "the right" did not fit on the right.

After locating an entry, proceed directly to the referenced page.

D

E

F

G

H

I

L

Q

R

S

Ordering Information for MetaWare Manuals

Copies of the *High C Language Reference Manual* may be ordered directory from MetaWare. The manual retails for $16.95 and is available at an educational discounted price of $12.95.

If your system includes the Professional Pascal TM compiler, you may want the three-manual set, including the programmer's guide, primer, and language extensions manual. This set retails for $32.95 and is available at an educational discounted price of $24.95. The manual set may also be ordered from MetaWare.

These prices include mail/shipping costs. California residents please add 6.5% sales tax. Please send: (1) an indication of educational affiliation, if appropriate, and (2) a check, money order, or written authorization to charge to your MasterCard or VISA account, with account number and expiration date to:

MetaWare Incorporated
903 Pacific Avenue, Suite 201
Santa Cruz, CA 95060-4429
(408) 429-META (=6382)

This page intentionally left blank.

The X Window System

ABSTRACT

This paper describes the X Window System, Version 11, supported by IBM/4.3. It is divided into two parts. The first part, the X User's Guide, is intended for people who use X with IBM/4.3. The second part, the X Programmer's Guide, is intended for programmers and system administrators who will configure, modify, and incorporate X into their application programs.

The **X User's Guide** contains the following chapters:

1. **Overview** describes the components of X.
2. **A Learning Guide for Using the X Window System** describes how to invoke and terminate X for a display, how to use the mouse and keyboard, and how to move, resize, and manipulate windows.
3. **Using X Applications** describes the xterm, xclock, and xload applications.

The **X Programmer's Guide** contains the following chapters:

4. **Utilities** describes each of the utilities provided with X.
5. **Customizing X** describes how to change default window characteristics, and how to configure X for particular environments.
6. **Customizing Uwm** describes how to modify the programmable window manager available with X.
7. **The Bitmap Editor** describes how to use X's editor for creating and editing a bitmap.

The following appendices appear at the end of the paper:

A. **X Colors**

B. **X Fonts**

C. **ASCII Code**

D. **Xterm Escape Sequences**

The X User's Guide

1. **Overview** describes the components of X.
2. **A Learning Guide for Using the X Window System** describes how to invoke and terminate X for a display, how to use the mouse and keyboard, and how to move, resize, and manipulate windows.
3. **Using X Applications** describes the xterm, xclock, xload, and xfd windows.

1. Overview

In X, a "display" is a server that manages one or more physical devices (called "screens") on which computer output appears. Further, X lets you divide each screen into multiple"windows." A "window" is a rectangular region on the screen that performs the functions normally associated with the entire screen.

Further, more than one display (server) can be active on one workstation at a time. The result, then, is a hierarchy, with multiple displays managing multiple screens, on each of which multiple windows appear. It is important to understand this usage of the word "display," as it permeates the remainder of this article.

X's network transparency allows applications that reside on one workstation to run on other workstations with screens of the same or different model.

1.1. Components of the X System

To use X requires the components listed below. The rest of this chapter describes these components.

- The IBM Academic Operating System 4.3
- Hardware
- User Interface
- The X Server
- X Applications

1.2. The IBM Academic Operating System 4.3

X runs under the IBM Academic Operating System 4.3. The user should be familiar with a UNIX operating system before using X. All UNIX operations can be executed from the X terminal emulator window, called the "xterm window."

1.3. Hardware

To use X, you must have a workstation equipped with a keyboard, a mouse, and one or more of the following devices. X supports the following workstations and devices, plus a standard keyboard and mouse:

- The IBM RT PC with up to three of the following:
  - The IBM 6155 Extended Monochrome Graphics Display
  - The IBM Academic Information Systems experimental display (which is no longer available)
  - The IBM 5081 Megapel Display
- The IBM 6152 Academic System with either or both of the following:
  - The IBM Video Graphics Array display adapter and displays
  - The IBM 8514-A Display

1.4. User Interface

X receives user input from three sources: the network, the keyboard and the mouse. Use the keyboard to enter commands and edit files. Use the mouse to move to another window, invoke menus, and select menu options.

You can also use the mouse in X application programs, to scroll the contents of a window, draw a picture, request information from a specific region of a window, or cut and paste information.

When a workstation has more than one screen running the same X server, moving the mouse can cause its cursor to jump from screen to screen.

1.5. The X Server

The X server interprets all X requests and monitors all display activity. The X server provides a library of fonts and colors that can be used in application windows, and to enhance the appearance of information.

1.6. X Applications

Most X applications "open," or present, their own windows. The following table is a list of supported X applications by type.

APPLICATION TYPE	NAME	FUNCTION
Operating System Interface	xterm	emulates DEC VT102 terminal
Monitor	xclock	displays digital or analog clock
	xload	displays system load average
Utility	xfd	displays font characters
	xhost	accesses host
	xinit	initializes X from UNIX shell
	xlsfonts	lists all X fonts
	xrdb	establishes X defaults
	xrefresh	redraws entire screen
	xset	sets X environment parameters
	xsetroot	sets root window characteristics
	xwd	dumps a window into a file
	xwininfo	accesses system information for a window
	xwud	undumps a window from a file
Window Manager	uwm	manages windows
Graphics Editor	bitmap	edits bitmap files
Sample	ico	displays icosohedron in motion
	muncher	displays random pattern generator
	paint	paints and prints a simple picture
	plaid	generates a random plaid

Descriptions of these applications appear later in this paper.

2. A Learning Guide for Using the X Window System

This chapter includes the following topics:

- starting and ending X
- characteristics of an X display
- managing windows
- opening windows

2.1. Starting X

There are two ways X may be configured for startup:

- With X running continuously on the display
- With X invoked by a command from the UNIX shell

2.1.1. With X Running Continuously

If X is running continuously, you will see a mouse cursor and a terminal window with a "login:" line.

(1) Ensure that the mouse cursor is in the login window by moving the mouse if necessary.

(2) Type:

loginID **< Enter >**

where *loginID* is your login ID.

(3) Type:

password **< Enter >**

where *password* is your password. If X normally runs continuously, you will not be able to proceed with the learning exercises.

2.1.2. With X Invoked by a Command

If the X window system is not running continuously, you have to enter a command to start it. To facilitate your learning of how X works, IBM has provided a command in the directory /usr/guest/guest/xwindows for you to use with the following material.

(1) To begin, ensure that you have already logged into a UNIX shell (by entering your login ID and password).

(2) Type:

set path = (/usr/guest/guest/xwindows $PATH) < Enter >
xwindows [*:display*] [*screen . . .*] < Enter >

The values for *display* are :0 through :7, with :0 the default.

Note: More information on the *xwindows* command can be obtained by typing:

man xwindows < Enter >

Simply typing **xwindows** starts a single display managing a single screen. X searches for an available screen, using the following order:

8514, vga, mpel, ega, apa16, aed

where:

8514 is the IBM 8514 PS/2 Color Graphics Display Adapter 8514/A
vga is the IBM Video Graphics Array (VGA) Display
mpel is the IBM 5081 Display with MegaPel adapter
ega is the IBM 5154 Enhanced Color Display with adapter
apa16 is the IBM 6155 Extended Monochrome Graphics Display
aed is the IBM Academic Information Systems Experimental Display

On the RT, by default the mouse cursor moves from the IBM 5081 to the IBM 5154 to the IBM 6155 to the aed. On the IBM 6152, by default the mouse cursor moves from the IBM 8514 to the VGA.

To start a display with more than one screen, include screen names on the command line. The sequence in which you enter the screen names, left to right, overrides the default mouse cursor movement. By using screen names you can ensure that the mouse cursor will move smoothly from left to right across your screens in the order that they reside on your display table.

If you do not start an available screen when you start X, you can do so later by issuing the *xwindows* command with the appropriate screen name. Note, however, that starting X separately for a screen invokes a separate copy of the display (server) and prevents cursor travel between groups of screens.

2.1.3. X Initialization

As the display initializes, the background (known as the "root window") and its cursor appear on each screen in the group. After a beep, an analog clock appears in the lower right corner of each screen of the group. Then two xterm windows appear, one atop the other, in the upper left corner of each screen. The xterm window which is darker than all the rest is the console window. All system messages appear in it. In the lighter xterm windows, the UNIX c-shell is running.

2.1.4. Console Focus

Console focus identifies the display that receives keyboard and mouse input. Ensure that the console focus is set to the display on which the console window appears. If it is not, the mouse does not move and keyboard input is not accepted. To change the console focus, press the <Alt> and <Scroll Lock> keys simultaneously.

2.1.5. Accessing Help

X includes an on-line help system with information about commands, window manager functions, and default keybindings. To access the Help Menu and select its options, proceed as follows:

(1) Move the mouse cursor to the root window.

(2) Hold down the right-most mouse button. The Main Menu will appear.

(3) Move the mouse to highlight the Help option on the Main Menu.

(4) Release the mouse button. Again, hold down the right-most mouse button. The Help Menu will appear. Select an option by moving the mouse cursor to the option and then releasing the right mouse button. The options operate as follows.

Help on Topic: When you select this option, another menu appears. Select one of the topics, and the available help text will appear.

Help on Keybindings: A help list appears that describes how to manage windows by keyboard and mouse.

X Command Summary: A help list appears that summarizes X applications and their associated commands.

2.1.5.1. Menu Help

Function menus are those that can be chosen from the Main Menu. Each function menu includes a Help option. Choose the Help option to view help information describing the selections on that menu.

2.1.5.2. UNIX Help

UNIX man pages provide information on UNIX commands. You can view them from the UNIX shell (an xterm window) by typing:

man *commandname* **< Enter >**

where *commandname* is the UNIX command name.

2.1.6. Ending X

You can end X from the Exit Menu or from the console xterm window.

2.1.6.1. Ending X from the Exit Menu

To end X from the Exit menu, do the following:

(1) Move the cursor to the display where the console xterm was started. (Remember, the console xterm is darker than the rest of the xterm windows.)

(2) Make sure the focus is on the display containing the console xterm. If it is not, press < Alt > and < Scroll Lock > simultaneously to change the focus.

(3) With the cursor resting in the root window, hold down the right mouse button. The Main Menu will appear.

(4) Move the mouse cursor to highlight the Exit option.

(5) Release the mouse button. Again, hold down the right mouse button. The Exit Menu will appear.

(6) Move the mouse to the Exit X option.

(7) Release the right mouse button.

If X was started with a login window, all user-created windows disappear. A login window with the "login:" prompt reappears.

If X was started with the *xwindows* command, all windows disappear and the UNIX shell display reappears.

If nothing happens, the cursor may not have been in the console xterm window, or the focus may not have been on the display containing the console window. Carefully repeat the above steps.

2.1.6.2. Ending X from the Console Xterm Window

To end X from the console xterm window, do the following:

(1) Move the mouse into the console xterm window.

(2) Type:

exit < Enter >

If X was started with a login window, all user-created windows disappear. A login window with the "login:" prompt reappears.

If X was started with the *xwindows* command, all windows disappear and the UNIX shell display reappears.

2.1.6.3. When All Else Fails

If the console window is not receiving input, and if the menus are not working, you can end X with UNIX commands. Enter these commands from an xterm window that is running on that X server or from a display that is logged into the workstation on which X is running.

(1) Type:

ps aux | grep Xibm <Enter>

The process id (PID) and other information about the Xibm process will appear. Find the number in the second column of the displayed information for the Xibm process (not the *grep Xibm* process). This is the Xibm PID.

(2) Type:

kill -9 *pid* <Enter>

where *pid* is the Xibm PID located in the previous step.

2.1.6.4. An X Startup Command Example

In the directory /usr/skel is a shell script named *xwin*. You can copy it to your home directory and execute it. Xwin starts Xibm and one xterm window. Experienced users will probably want to modify their copies of xwin to suit personal tastes.

2.2. Characteristics of an X Display

This section provides a brief introduction to an X display. You might want to have X running so you can experiment with the mouse and keyboard.

2.2.1. When X First Comes Up

When an X work session (using the system defaults) begins, the following appear on the display:

- the background, known as the root window
- the mouse cursor
- two xterm windows
- an analog xclock window

The next sections explain these items further.

2.2.2. The Mouse Cursor and Mouse

Generally, as you move the mouse, you make the mouse cursor on the display move in the same direction at the same relative speed. This section describes two special features of the mouse.

2.2.2.1. Movement Across Screens

When the X server is running more than one display, and the mouse cursor moves off the right edge of one display, it appears on the left edge of another display. (Which display is determined either by default or by the order in which the displays were invoked. See "Starting X" earlier in this chapter.) If the cursor moves off the

right edge of the last display, it reappears on the left edge of the first display. Similar movement occurs from left to right.

2.2.2.2. Cursor Shapes

As the mouse cursor enters different windows on the screen, its shape may change. The cursor resembles an "X" when it is in the root window. When the cursor enters the xterm window, it becomes an "I." An X application may define several mouse cursor shapes for use in its window. For example, when the cursor moves into the scrollbar region of the xterm window, it is changes from an "I" to a double-headed arrow. The mouse cursor shape serves as a visual indicator of the foreground process within a window. The foreground process is the process that receives and acts on mouse and keyboard input.

Information about specific cursor shapes and functions appears in later chapters that describe specific X applications.

2.2.3. Windows

This section describes several types of windows and window presentation.

2.2.3.1. The Root Window

The "root window" is the display background, whose default pattern is a gray zigzag. The X server is the application that runs in the root window. All X application windows are built on top of the root window. The root window also owns the X cursor. To change the appearance of the background and cursor, see the *xsetroot*(1) man page.

2.2.3.2. X Application Windows

Some X applications create their own windows; others operate within existing windows. Each X application window can differ from others in function, size, use of color and fonts, cursor function and shape, and so forth.

2.2.3.3. Window Layering

Each window on the display exists on its own "layer." X assigns the layer level chronologically. That is, the first window to appear is on the bottom layer. The next window is on top of the first one. If another window appears, it will occupy the top layer. The group of layered windows is called the "window stack."

2.2.3.4. Overlapping vs. Tiled Window Management

When a window manager allows layered windows to overlap one another, it is known as a "overlapping" window manager. When a window manager prohibits windows from overlapping one another, but instead automatically resizes windows to fit on the display without overlap, the window manager is known as a "tiled window manager." X provides a sophisticated overlapping window manager which allows you to alter the layering order. To do so, you use the Top of Stack and Bottom of Stack options on the Manage a Window Menu (discussed later in this chapter).

2.2.3.5. The Focus Window

The "focus window" is the window that receives all keyboard and mouse input. X is configured so that the window containing the cursor is the focus window. In X, this is called "real estate mode." The focus may be changed so that a selected window is the focus window regardless of the cursor position. In X, this is called

"listener mode."

Listener mode (separating the focus window from the cursor window) applies only to the display on which it was requested. If the same X server is running on two displays, one may be in listener mode and the other in real estate mode. When the mouse moves from one display to another, the mode in effect on that display takes

effect. The mode does not transfer to the next display with mouse movement. (Changing between real estate mode and listener mode is described later in this paper.)

2.2.3.6. Icons

"Icons" are symbols that represent larger items or actions. In X, a window icon represents a window. You can change a window into an icon and back again, so that the window remains readily accessible, but it appears in full size only when needed.

Only the window manager can issue commands to window icons. They cannot receive commands from the process running in the window. You can start a process in a window and then reduce the window to an icon while the process is running. You cannot provide additional keyboard input to that window until you change it from an icon back to a window.

Note: Remember, icons cannot receive keyboard input. Therefore, if you move the cursor to an icon and begin to type, X interprets this to mean that you want to *change the name* of the icon.

2.3. Managing Windows

The X window manager (uwm) gives you the ability to move and resize windows, change them to or from icons, and shuffle overlapped windows to the top of the window stack. It also provides more advanced functions, such as freezing and unfreezing the display, creating and exiting a window, and accessing help text.

The window manager includes menus from which you invoke window management functions. You can also perform frequently-used functions with keyboard/mouse actions.

2.3.1. The Menus

The six window management menus are:

- Main Menu
- Create a Window
- Manage a Window
- Manage the Display
- Hosts
- Help
- Exit

Each screen has its own copy of the window manager, even if the screens are running on the same X server. Therefore, window management functions apply only to the screen on which they appear.

2.3.1.1. Using the Menus

Use the Main Menu to select one of the function menus. You learned earlier in this chapter how to access the Help Menu. You access other function menus in the same way:

(1) With the cursor in the root window, hold down the right-most mouse button. The Main Menu appears.

(2) Move the cursor to highlight one of the menu selections.

(3) Release the mouse button. Again, hold down the right mouse button. The selected function menu will appear. Do not release the mouse button. You

must hold it down until you make a selection from the function menu.

(4) Move the cursor on the function menu to highlight the desired selection.

(5) Release the mouse button to make the selection.

2.3.1.2. The Help Option

Each function menu has a Help option. Choosing this option retrieves a help window describing the function menu selections.

2.3.1.3. Create a Window Menu

The Create a Window Menu enables you to create, or "open," five different types of windows:

- Xterm
- Xclock Analog Clock
- Xclock Digital Clock
- System Load
- Console Xterm

When you select one of these options, the menu disappears and a special right angle cursor appears. The right angle cursor represents the upper left corner of the window. A dimension box also appears in the upper left corner of the display. To cause the selected window to appear, do the following:

(1) Move the cursor to the desired position.

(2) Click the right-hand mouse button. The window will appear.

2.3.1.4. Manage a Window Menu

The Manage a Window Menu lists the six operations that can be performed on a window or icon:

- Move
- Resize
- (De)Iconify
- Top of Stack
- Bottom of Stack
- Close a Window

When you select one of these options, a special doughnut cursor appears. To proceed with the operation, first move the cursor into the window where the action is to occur. Be careful not to press a mouse button as you move the mouse; if you do, the action is canceled.

2.3.1.4.1. Move

(1) As you hold down the right-most mouse button, move the mouse. An outline of the window moves as the mouse cursor moves. This outline shows the position the window will assume when you release the button.

(2) Release the button to complete the move.

2.3.1.4.2. Resize

The position of the doughnut cursor in the window determines how the window will be resized.

(1) To move a border, place the cursor just inside the center point of the border. To move a corner, place the cursor just inside the corner.

(2) Hold down the right mouse button. The resize dimensions appear in the upper left corner of the window.

(3) Move the mouse out of the window to expand its size. Move the mouse within the window to contract its size. The new size of the window is indicated in the resize dimension box. For xterm windows the resize dimensions are expressed in number of characters. For all other windows, they are expressed in number of pixels. An outline also indicates the size the window will assume when you release the button.

(4) Release the mouse button to resize the window.

Note: Xterm windows do not resize their character fonts. That is, the characters within the xterm window do not grow or diminish to match the resized window.

2.3.1.4.3. (De)Iconify

This option converts a window to or from an icon. As the conversion occurs, X may move the icon/window to a more suitable location.

(1) To initiate the conversion, move the cursor to the appropriate window or icon.

(2) Press and release the right mouse button.

(3) To move the window/icon during the conversion, hold down the right mouse button. Then move the mouse to the new location. An outline shows the position the window/icon will assume when you release the button.

(4) Release the button to complete the conversion.

2.3.1.4.4. Top of Stack

You cannot move a window to the top of the stack if it is completely hidden beneath another window. If this is the case, first move the overlaying window, or use the Send Bottom Window to Top option from the Manage the Display Menu.

Move the mouse to the selected window and click the right button. The window moves to the top of the stack.

2.3.1.4.5. Bottom of Stack

You cannot move a window to the bottom of the stack if it is completely hidden beneath another window. If this is the case, first move the overlaying window.

Move the mouse to the selected window and click the right button. The window moves to the bottom of the stack.

2.3.1.4.6. Close a Window

Move the cursor to the selected window and click the right mouse button. The selected window disappears.

2.3.1.5. Manage the Display Menu

This menu lists seven display operations:

- Focus Select
- Refresh Display
- Freeze Display
- Unfreeze Display
- Top Window to Bottom
- Bottom Window to Top
- Restart uwm

With the Refresh option, you needn't move the cursor. However, with any other option, the special doughnut cursor appears. You must move the doughnut cursor out of the menu. As you use the mouse to move the cursor, be careful not to press a mouse button; if you do, the action is canceled.

Only the Focus Select option requires that the cursor be in a specific window. For all other options, place the doughnut cursor anywhere on the display outside the menu. When you have moved the cursor, click the right-most button to begin the selected operation.

2.3.1.5.1. Focus Select

The Focus Select option toggles listener mode on and off. When listener mode is on, all keyboard and mouse input is directed to one window, regardless of the cursor position. For more information on listener mode, refer to "The Focus Window" earlier in this chapter.

To toggle listener mode on or off, first choose the Focus Select option. Then move the cursor to the desired focus window and click the mouse button.

2.3.1.5.2. Refresh

The Refresh option redraws the entire display. Just select the option to cause the refresh.

2.3.1.5.3. Freeze Display

The Freeze Display option withholds input from the display. The display is not updated as long as it is frozen. If the Refresh option is chosen after the display is frozen, all the windows are rewritten, but their contents is missing; the contents of the display buffer is not kept current through updates. Also, if windows are moved after the display is frozen, portions of windows may be blank.

The Freeze Display option is useful to capture a particular display state before dumping it to a printer with the UNIX *bitprt* command.

2.3.1.5.4. Unfreeze Display

The Unfreeze Display option undoes the Freeze Display option, restoring the display to its normal operation.

2.3.1.5.5. Send Top Window to Bottom

This option sends the window on the top of the display stack to the bottom. For a description of X display layering, see "Window Layering" earlier in this chapter.

2.3.1.5.6. Send Bottom Window to Top

This option sends the window on the bottom of the display stack to the top. For a description of X display layering, see "Window Layering" earlier in this

chapter.

2.3.1.5.7. Restart uwm

This option kills the current window manager and restarts another based on the $HOME/.uwmrc file. This enables the quick implementation of just-completed modifications to the $HOME/.uwmrc file. (See the chapter entitled "Customizing Uwm" later in this paper.)

2.3.1.6. Hosts Menu

You use the Hosts Menu to perform a remote login to another workstation or machine on the network. To do so, follow these steps:

(1) Choose the Hosts option from the Hosts Menu. A window appears with the *Enter host name:* prompt.

(2) Type:

machinename **< Enter >**

A window appears on an xterm display logged into that machine.

2.3.1.6.1. Exit

The Exit Menu includes two exit options:

- Exit uwm
- Exit X

Neither option requires your further action after selection. A description of each option follows. If for some reason the menu system becomes inaccessible, there is a way to exit from an xterm window. (See "When All Else Fails" earlier in this chapter.)

2.3.1.6.2. Exit uwm

This option kills the window manager. Window manager menus cease being accessible, and the mouse and keyboard cease managing windows when you select this option.

You may start a new window manager by typing the following command in an xterm window:

uwm & < Enter >

2.3.1.6.3. Exit X

This option exits X. If X begins with a login window, this option logs you out of the working session. If you started X with the *xwindows* command, all X windows disappear and the UNIX shell display reappears.

2.3.2. UNIX Commands

The X server and each of the applications it runs are treated as a single process by the UNIX system. Therefore, you can use the following UNIX commands to control the X process:

COMMAND	DESCRIPTION
command &	initiates the named process in the background
bg	moves stopped foreground process to background
fg	moves newest background process to the foreground
kill -9 %*x*	terminates the background process identified by job number *x* (from the *jobs -l* command)
kill -9 *pid*	terminates the background process identified by process number *pid* (from the *ps -aux* command)
^C	terminates a foreground process
^Z	stops a foreground process

For more information on these commands, see the respective UNIX man page.

2.3.3. Keybindings

The most frequently-used window management functions have been bound to keys and mouse clicks. These functions and their keybindings and mouse movements are as follows:

FUNCTION	KEY	MOUSE BUTTON	MOVES MOUSE?
Move	ALT	right	yes
Resize	ALT	both	yes
(De)iconify	ALT	left	yes
Top of Stack	Control	right	no
Bottom of Stack	Control	left	no

Move the mouse to the selected window before performing these actions. Press the key and mouse button in unison. When a function moves the mouse, the function is completed after you release the button.

This example describes how to move an xterm window.

(1) Place the mouse cursor in the xterm window.

(2) Hold down the right mouse button and the ALT key.

(3) Move the mouse to the desired location.

(4) Release the mouse button and ALT key.

These keybindings also appear under the *Help on Keybindings* option of the Help Menu.

2.3.4. Programming The Window Manager

The X window manager is programmable, allowing you to change keybindings and menus to suit your needs. You should be familiar with the default configuration of uwm before programming your own. Information on the default configuration appears in the chapter entitled "Customizing X." Information on programming uwm appears in the chapter entitled "Customizing Uwm."

2.4. Opening Windows

Several X applications open their own windows. This section describes the command options common to these application windows (color, fonts, window positioning).

The five X applications that open windows are:

- bitmap
- xclock
- xfd
- xload
- xterm

Typing the name of the application in an xterm window causes the new window to open and the application to begin. (With bitmap, you must also supply the name of the bitmap file.) Usually you run these applications in the background. Otherwise, no other command can be executed in the xterm window until the application terminates. Adding an ampersand (&) to the end of the command line causes the process to run in the background. For example, **xclock &** begins the application, opens an xclock window, and frees the xterm window for other commands.

There is one advantage to running an application in the foreground: you can terminate it in an instant by typing **^C** in the xterm window.

2.4.1. Positioning a Window in Uwm

To position a window in a specific area of the screen, you enter "offsets" on the command line that invokes the application. When uwm is running, if you don't enter offsets, the window does not automatically appear on the screen. Instead, the special right angle cursor and the dimensions box appear. To make the window appear, first place the right angle cursor where the upper left corner of the window is to be. Then click either mouse button. The window appears with its upper left corner positioned at the cursor.

2.4.2. X Application Options

If you simply enter the application name on the command line, X uses default settings to format the application window. You can override the default settings by adding the following options to the command line. (**Note:** Bitmap does not use any of these options.)

bd (border color)

bg (background color)

fg (foreground color)

bw (border width)

fn (font); not used by xfd

rv (reverse video)

2.4.2.1. Colors

The "background color" is the window color on which all text and graphics are drawn. The "foreground color" is the color used for text and graphics. The "border color" is the color of the window frame. Appendix A lists available colors. The default colors are:

AREA	DEFAULT COLOR
background	white
foreground	black
border	black

To change the colors used by a window, type:

Xapplication -bg background -fg foreground -bd border & <Enter>

For example, to execute xload with an aquamarine foreground, a coral background, and a light blue border, enter the following command in the xterm window:

xload -fg aquamarine -bg coral -bd LightBlue & <Enter>

On a monochrome display, X ignores color values other than "black" and "white." If you enter other values, X substitutes the defaults.

2.4.2.2. Fonts

X includes an extensive library of fonts. Appendix B lists the font names. For the following X applications, the font option (−fn) causes the named information to appear in the selected font:

- xclock: digital clock time and date
- xfd: verbose mode information (Note: uses the −bf rather than the −fn option)
- xload: workstation name
- xterm: command entry and status messages

Note: You must choose a fixed font in the xterm window. In other windows, you may choose either fixed or proportional fonts.

The following command invokes xterm with an fg-13 font:

xterm -fn fg-13 & <Enter>

2.4.2.3. Reverse Video

The **-rv** option reverses the foreground and background colors. On a monochrome display, the background becomes black and the foreground becomes white.

2.4.2.4. Border Width

Border width is expressed in pixels. The width does not impinge on window dimensions, but is added onto the outside of the window. The command **xload -bw 25 &** creates an xload window with a border that extends 25 pixels from each window boundary.

2.4.2.5. host:display.screen

An X application may open a window on a workstation, server, or display other than the one from which the application was started.

If the server is running on more than one display on a given workstation, each display is assigned a number. The assigned number follows the order these displays were listed on the *xwindows* command line. The leftmost display on the command line is 0, the next is 1, and so on. Unless specified by the Xibm command (discussed in a later chapter of this paper), the first server started on a workstation is 0,

the second server is 1, and so on.

For example, if you enter the command:

xclock rook:2.1 & < Enter >

in an xterm window on the bishop workstation, you will start xclock on the rook workstation's third server and that server's second display. (Remember, servers and displays are numbered beginning with 0.)

2.4.2.6. Geometry

You can give specific dimensions to a window and place it anywhere on the screen by using the geometry option. This option takes the form:

= wxh ± xoff ± yoff

where:

= wxh is the width and height of the window

± xoff is the pixel offset in the x direction

± yoff is the pixel offset in the y direction

The width and height are expressed in pixels for all windows except xterm. For xterm windows, width and height are expressed in characters. No blank spaces are allowed between the parameters of this option.

There are four offset origins as summarized in the following table. In each of these coordinate systems, **x** is the horizontal axis and **y** is the vertical axis.

ORIGIN	SCREEN CORNER	WINDOW CORNER COUNTED TO
+0+0	upper left	upper left
+0-0	lower left	lower left
-0+0	upper right	upper right
-0-0	lower right	lower right

You need not enter the offset parameters. You can enter **= wxh ± xoff ± yoff** by itself. To use the offset when uwm is running, the **= wxh** portion of the command must be specified. If uwm is not running, you can enter offsets without specifying width and height, but the equals sign must precede the offsets. Both offset parameters must be entered.

For example, the command:

xload -bw 25 = 300x200-30+30 & < Enter >

places an xload window with a border of 25 pixels slightly inside the upper right corner of the display. Because the offset is counted from the corner of the window, *not the border*, the 25-pixel border rests 5 pixels within the upper right borders of the root window.

The command

xload -bw 25 = 300x200 & < Enter >

creates a 300x200 xload window, but when uwm is running it will not automatically appear. You must use the mouse to position the window. (See the next section.)

The command

xload -bw 25 =-30+30 & <Enter>

places the xload window slightly within the upper right corner of the display when uwm is not running. The window will assume its default size. If uwm is running, you must use the mouse to position the window. It will be of default dimensions.

3. Using X Applications

This chapter describes in detail how to use xterm, xclock, and xload and their windows.

3.1. Xterm

Xterm emulates a DEC VT102 terminal, providing a consistent interface to the UNIX operating system regardless of the configuration of your workstation. You can use the xterm window to enter UNIX and X commands, including those for UNIX editors such as emacs, vi, and ed, and those to compile and run programs. You can use all DEC VT102 escape sequences. (See Appendix D for a list of these.)

The default xterm window provides cut, paste, and menu facilities. You can also request a scrollbar and a log file to record keystrokes executed in the xterm window.

The rest of this section provides summary information on xterm options, and on using scrollbars, cut and paste, and menus.

3.1.1. The Xterm Command Options

The xterm command options pertaining to colors and font are described in the preceding chapter. Some special xterm options appear in this section. For a complete discussion of the command, see the *xterm* man page.

Most of the options require a leading hyphen (−) or plus sign (+). The hyphen activates the option. The plus sign returns the option to its default setting.

-132 This option enables the xterm window to switch between 80-column (the default) and 132-column mode. Once enabled, the switch occurs when the xterm window receives the following escape sequences:

- To switch from 80 to 132: Esc [? 3 h
- To switch from 132 to 80: Esc [? 3 l

where "l" is a lower case L. Note that typing the escape key (Esc) produces a ^[on the display. You can use *echo* to send these escape sequences to xterm, enclosing the strings in double quotes thus:

echo "Esc[?3h" < Enter >

or

echo "Esc[?3l" < Enter >

-b *pixels*
This option sets the size of the inner border (the space between the inner edge of the xterm character display and the xterm window border). The default is one pixel.

-C This option sends messages for /dev/console to the xterm window. It effectively creates a console xterm window.

-cr *color*
This option determines the color of the highlighted text cursor. The default is the foreground color. If no foreground color is specified or if the display is monochrome, it defaults to black.

-cu Because a bug exists in the *curses*(3x) cursor motion package, this option is necessary for programs using *curses* to interact correctly with the DEC VT102 terminal.

For example, *more*(1) uses *curses*. If **-cu** is not specified when *more* is running, leading tabs may intermittently disappear.

-e *command*
This option dedicates the xterm window to the specified command. The command can take arguments in the normal fashion.

Note: The *-e* option must appear after all other options on the command line. The xterm window vanishes after the specified command terminates.

-fb *font*
Xterm writes all bold characters in the xterm window in the font specified in this option. By default, bold characters are written as an overstrike of the font specified by the **-fn** option. The font specified by this option must be of the same point size as the the font specified by the **-fn** option. If the **-fn** option is not specified, the font specified by this option is the normal font and there is no bold font.

-i This option causes xterm to be an icon when it first appears on the display. By default, the icon appears directly beneath the mouse cursor when the application begins.

-j This option sets xterm to "jump scroll," to scroll more than one line at a time. Xterm defaults to jump scroll.

-l This option sets logging on. Logging causes every keystroke entered in the xterm window to be recorded in a file. The default file name is *XtermLog.XXXXX*, where *XXXXX* is the process ID of the xterm window.

Xterm creates the log file in the directory from which xterm was started. If the xterm window is a login xterm (either option **-ls** or option **-L** was specified), the log file appears in the home directory.

-lf *filename*
This option overrides the default file name for the log file.

-ls This option causes the xterm window to run under the shell specified in the .login file. Xterm reads the .login file and comes up in the home directory. This option is not used when the xterm window is opened using the **xinit** command in the /etc/ttys file. In that case, the **-L** option is used.

-L The **-L** option creates a login window when X is initialized using the **xinit** command in the /etc/ttys file.

-mb This option turns on the right margin bell. The bell rings whenever the cursor reaches the specified margin bell setting. This setting is established with the **-nb** option. The bell may be set to visual rather than audio using the **-vb** option.

-ms *color*
This options sets the color for the mouse cursor when it is in the xterm window. The default is the foreground color.

-n *windowname*
This option specifies the name of the xterm window. This name appears in the icon and dimensions box for the window, and is noted when **xwininfo** is run on the window. The default name is *xterm*.

-nb *number*
This is the location, expressed as the number of spaces left of the right margin, at which the margin bell rings. The margin bell is activated using the **-mb** option.

-rw This option turns on reverse wraparound mode. In reverse wraparound mode, the backspace key can move the cursor through the left margin and up to the end of the previous line. The cursor does not wrap back through the prompt.

-s This option disables synchronous scroll. The display is not continually updated and current keystrokes do not appear as they are typed, but the terminal executes commands much more swiftly. This may be useful when network latencies are very high, as when using xterm across a very large internet.

-sb This option produces a scrollbar at the left border of the xterm window. When the cursor enters this area, it changes to the special double arrow, indicating that mouse clicks will scroll the contents of the window backwards and forwards. The number of lines available for scrolling is normally 64. To change this default, use the **-sl** option.

-si With the scrollbar enabled, this option prevents the xterm window from scrolling to the bottom when keyboard or system input is received.

-sl *number*
This option sets the number of lines that are saved "above" the top of the window when the scrollbar is activate. The default is 64 lines.

-vb This option transforms the audio bell into a visual bell. The visual bell flashes the entire window in reverse video.

#±xoff±yoff
This option sets icon geometry. Icon geometry determines the position the icon assumes when the window first appears (using the **-i** option), or when the window is changed to an icon. (The icon does not occupy this position when the icon results from your selecting the *(De)Iconify* option on the *Manage a Window* menu.)

3.1.2. Menus

There are two xterm menus:

- xterm X11
- Modes

Both menus use a line to divide their menu options into two groups. The top group are toggles for command line options. If one of these selections is on, a check mark appears to the left of the selection.

The bottom group are commands. They provide a quick way to execute certain functions, such as closing the xterm window, resetting the xterm window, and sending signals to the application running in xterm.

3.1.2.1. Using Menus

To view the xterm X11 menu, hold down both the Shift key and the Control key. Then press the left mouse button. The top of the menu will appear at the cursor position. Release the Shift and Control key but do not release the left mouse button.

To view the Modes menu, hold down both the Shift key and the Control key. Then press both mouse buttons. The top of the menu will appear at the cursor position. Release both keys but do not release the mouse buttons.

To choose an option from either menu, move the cursor so it highlights the option. Release the mouse button(s) to make your choice.

3.1.2.2. The Xterm X11 Menu

The following table describes each xterm X11 menu selection, noting the default settings.

SELECTION	DEFAULT	FUNCTION
Visual Bell	off	Turns on the visual bell; same as **-vb** option
Logging	off	Turns on logging; same as **-l** option
Redraw	--	Refreshes the xterm window
Continue	--	Same as UNIX **fg** (SIGCONT)
Suspend	--	Same as UNIX **^Z** (SIGTSTP)
Interrupt	--	Same as UNIX **^C** (SIGINT)
Hangup	--	Closes the X window (SIGHUP)
Terminate	--	Closes the X window (SIGTERM)
Kill	--	Closes the X window (SIGKILL)

3.1.2.3. The Modes Menu

The following table describes each *Modes* menu selection, noting the default settings.

SELECTION	DEFAULT	FUNCTION
Jump Scroll	on	scrolls more than one line at a time; same as **-j**
Reverse Video	off	reverses foreground and background color; same as **-rv**
Auto Wraparound	on	wraps long entries to next line
Reverse Wraparound	off	backspace key can move through left margin to end of previous line; same as **-rw**
Auto Linefeed	off	insert extra linefeed
Application Cursors	off	enables use of arrow cursors
Application Pad	off	enables use of numerical keypad
Auto Repeat	on	holding key down produces multiple characters on display
Scrollbar	off	produces scrollbar at left border; same as **-sb**
Scroll to Bottom on Key	on	same as **-sk**
Scroll to Bottom on Input	off	same as **-si**
80 <-> 132 Columns	off	enables window to switch between 80- and 132-column mode; same as **-132**
Curses Emulation	off	fixes bug in *curses*(3x); same as **-cu**
Margin Bell bell; same as **-mb** T}	off	turns on right margin
Tek Window Showing	off	displays/hides Tek window
Alternate Screen	off	not available

SELECTION	DEFAULT	FUNCTION
Soft Reset	--	reset scroll region
Full Reset	--	clear window, reset tabs to 8-column width, reset terminal modes such as wrap, smooth scroll
Select Tek Mode	--	not available
Hide VT Window	--	not available

3.1.3. The Scrollbar

You can enable the scrollbar either using the Modes menu or using options on the command line. (See the **-sb**, **-sl**, and **-si** options on the *xterm* man page.)

The highlighted region of the scrollbar represents the amount of text appearing in the window. The darker region represents lines scrolled off the window. Use the **-sl** opton to change the number of available scrolled lines. The default is 64. Use the scroll options on the Modes menu to control scrollbar actions. Note that when the mouse cursor enters the scrollbar area, it becomes a double-headed arrow.

With the cursor in the scrollbar area, proceed as follows to scroll text:

- **To scroll to a specific portion of text,** position the cursor at the desired text location. (Remember, the length of the scrollbar represents the amount of scrolled text.) Position the cursor at the desired text position. Click both mouse buttons. The cursor becomes a horizontal arrow to indicate that the selected lines will appear at the top of the window.
- **To scroll up,** click the left mouse button. The line of text at the cursor position now appears at the top of the window. The cursor becomes an up arrow to indicate that the lines are scrolling upward.
- **To scroll down,** click the right mouse button. The line of text at the top of the window will now appear at the cursor position. The cursor becomes a down arrow to indicate that lines are scrolling downward.

3.1.4. Cut and Paste

Xterm provides a cut and paste facility to copy text from one area of an xterm window to another or to copy between different xterm windows. You can use cut and paste in the shell to construct commands from various already-executed commands and to paste them on the current line for execution. You can also use cut and paste within the vi editor.

To cut and paste text, proceed as follows:

(1) Position the cursor at the beginning of the text to be copied.

(2) Hold down the left mouse button.

(3) Move the cursor to highlight completely the text to be copied.

(4) Release the left mouse button.

(5) If you need to change the amount of selected text:

a. Hold down the right mouse button.

b. Move the mouse to adjust the text to be copied.

c. Release the right mouse button.

(6) Position the mouse cursor in the window that is to receive the copied text.

(7) Position the text cursor at the location where the text is to be copied. **Note:** If a space is needed between existing text and copied text, you must place the cursor one space to the right of the existing text.

(8) Click both mouse buttons. The copied text is inserted after the text cursor. Note that copying text in vi automatically opens up insert mode; vi remains in insert mode after the paste operation.

3.1.5. Exiting Xterm

There are four ways to exit the xterm window:

- using the xterm x11 menu
- using the *Close a Window* option from the Manage a Window menu
- typing **exit** at the xterm prompt
- using the UNIX **kill** command

If you type **exit** in the console window, the console window disappears and the X server terminates. The console window must continually be displayed for the entire duration of the X work session.

3.2. Xclock

Xclock reads the UNIX clock and displays the current time. You can view the time on either an analog (face and hands) or digital clock.

3.2.1. The Xclock Command Options

Many of the xclock command options (those pertaining to colors and fonts) are described in the preceding chapter. Options unique to xclock appear in this section. For a complete discussion of the command, see the *xclock* man page.

-analog or -digital
: The **-analog** option (the default) causes time to appear on a clock face. The analog format does not include the date. The **-digital** option causes time and date to appear in digital format: *day date hr:min:sec year.*

-hl *color*
: This options sets the color of the analog clock hands. Black is the default.

-padding *pixels*
: This option specifies the distance in pixels from the time display to the inner edge of the xclock window border. The default in analog mode is 8 pixels; in digital, 10 pixels.

-update *seconds*
: This option sets the frequency (in seconds) with which the time display is updated. The default is once every 60 seconds. The second hand does not appear on the analog clock unless the display is updated at least every 30 seconds. To update the clock display every second, type the command as follows:

xclock -update 1 & <Enter>

Regardless of the update setting, xclock automatically updates the display every time it moves to the top of the window stack, and every time it changes from an icon to a window.

3.3. Xload

Xload monitors the workstation's system load average and displays it on a bar graph. This is the same value displayed by UNIX *uptime*.

Each scale line in the xload window is equivalent to one average process that is waiting for execution. On a workstation with low activity, no horizontal scale lines appear because the system load average is less than one process.

When the xload display appears on a different workstation using the **host:display:screen** option, it shows the system load average of the original workstation, not the one on which the display appears. The name of the monitored workstation automatically appears in the upper left corner of the window.

3.3.1. The Xload Command Options

Many of the xload command options (those pertaining to colors and fonts) are described in the preceding chapter. Options unique to xload appear in this section. For a complete discussion of the command, see the *xload* man page.

-hl *color*
: This option sets the color for the workstation name and scale lines.

-scale *n*
: This option sets the number of vertical graph divisions. Each division is a horizontal line across the window and marks one (average) process waiting for execution.

-update *seconds*
: This option sets the frequency (in seconds) with which the load display is updated. The minimum (and default) is once every 5 seconds.

  Regardless of the update setting, xload automatically updates the display every time it moves to the top of the window stack, and every time it changes from an icon to a window.

The X Programmer's Guide

4. **Utilities** describes each of the utilities provided with X.
5. **Customizing X** describes how to change X default window characteristics, and how to configure X for particular environments.
6. **Customizing Uwm** describes how to modify the programmable window manager available with X.
7. **The Bitmap Editor** describes how to use X's editor for creating and editing a bitmap.

4. Utilities

This chapter describes the utilities included with X:

- xfd
- xhost
- xlsfonts
- xrdb
- xrefresh
- xset
- xsetroot
- xwd
- xwininfo
- xwud

Enter these commands in the xterm window, or from a screen not running X. You need not type an ampersand at the end of the command when invoking one of these utilities. They execute immediately and will not inhibit further entries into the xterm window or UNIX shell.

4.1. Xfd

Xfd displays characters of a specified font. The location of the character on the screen corresponds to its ASCII and hexadecimal code value. In the default xfd window, the upper left box is decimal 0, hexadecimal 0x0. These numbers increment across the row and down the window. To view both the decimal and hexadecimal code for any character, move the mouse cursor to the character and click both mouse buttons.

X supports 8-bit fonts, which can include up to 256 characters. Xfd defaults to a 16x16 grid to display the characters in these fonts. If you resize the xfd window, the grid size changes accordingly. If the characters are small enough, the entire 16x16 xfd window fits on the display. Otherwise, only a portion of the window appears. To view the remaining characters, scroll the window by moving the mouse cursor into the xfd window and clicking the right and left mouse buttons respectively.

To display lower rows of the font display, use the **-start** option. This option specifies which character will be displayed in the upper left corner of the grid, thereby shifting the window's display focus.

You can close the xfd window by typing one of the following in the xfd window:

q
Q
^C

If the window is an icon, you must change it back to a window before closing it.

4.1.1. The Xfd Command Options

Many of the xfd command options (those pertaining to colors) are described in the preceding chapter. Options unique to xfd appear in this section. For a complete discussion of the command, see the *xfd* man page.

font Specify on the command line the simple name of the font you want displayed:

xfd *font*

The names of fonts available with X appears in Appendix B. If you omit this option, the default font "fixed" appears. You need not include a file name extension or path name for the font. If the font you specify is not in the /usr/lib/X11/fonts directory or has an extension other than .snf, then specify the full pathname and/or extension.

For example, if the font "jazzy.cnf" exists in the /special directory, use the following command to display it in an xfd window:

xfd /special/jazzy.cnf & < Enter >

Note: Use the following command to display the 25-point cyrillic font in the /usr/lib/X11/fonts directory:

xfd cyr-s25 & < Enter >

-bf *font*
This option selects the font used for character information that appears at the bottom of the display.

NOTE: If you choose a very large font, you may need to resize the window to read its information.

-gray This option highlights the empty region surrounding each character. This region is part of the character. Using this option causes the window background to be gray, the character to appear in the foreground color, and the empty region to appear in the background color.

-in ***iconname***
The **-in *iconname*** option sets the icon name to that specified by the **-icon** option. This name appears in the icon, overriding any name specified by the **-tl** option.

-start ***character#***
This option moves the specified character to the upper left box of the xfd grid. Other characters follow this one in their usual order. Use this option to access the lower rows of oversized fonts.

For example, if you choose character # 117 to appear in the upper left box, characters 117 through 255 would follow it in the default 16x16 grid. Characters 0 through 116 would fill in the remaining grid boxes.

-tl ***title***
This option sets the name of the window. This name appears in the uwm sizing box.

-verbose

This option causes additional information to appear when you move the mouse cursor to a character and click both buttons. Normally, just the ASCII and hexadecimal codes for the character appear. The additional information includes:

character width
left bearing
right bearing
ascent
descent

Information provided by this option appears at the bottom of the xfd window. Use the **-bf** ***font*** option to change the font in which this information appears.

4.2. Xhost

Xhost changes the list of hosts that can access the X server on the home workstation. The changed access privileges last for the duration of the current work session. When X is exited or when the user logs out of X, this access list is reset to that in the /etc/X*.hosts files. (The "*" in the /etc/X*.hosts file name is the display number for the workstation. For example, if the display number is 1, the file /etc/X1.hosts lists the hosts that have access privileges to the home workstation when display 1 is in use.

To grant a host permanent access privileges, you must either edit the /etc/X*.hosts file or modify the .login file.

4.2.1. Command Format

Use the following command to invoke xhost:

xhost ± ***host*** ± ***host*** . . .

You must execute this command on the home workstation. The host entries may be preceded by a +, −, or no sign, with the following effects:

+ or no sign
Grants access to the specified display for the named workstation(s).

− Denies access to the specified display for the named workstation(s).

To review a list of workstations with current access privileges to a display, simply type the **xhost** command (with no options). This list includes the name of the workstation, and uses information in the /etc/hosts file.

4.3. Xlsfonts

Use the *xlsfonts* command to list the name of the fonts in the /usr/lib/X11/fonts directory, or to see if a particular font exists in the directory.

4.3.1. Command Format

Use the following command to invoke xlsfonts:

xlsfonts ***pattern host:display:screen***

Use the **pattern** option to limit the list to fonts whose names match the pattern. The ? and * wildcard characters may be used in the pattern, thus:

? matches any single character

* matches any string of characters (including null)

If either of these wildcard characters is used, enclose the expression in single quotes. For example, the following command presents a list of all fonts whose names begin with *gothic*:

xlsfonts 'gothic*'

This produces the following list:

gothic.12
gothic.12.snf
gothic.15
gothic.15.snf

For information on the **host:display:screen** option see Chapter 2.

4.4. Xrdb

Use *xrdb* to set the contents of the .Xdefaults file. For more information on the defaults file, see Chapter 4.

4.5. Xrefresh

Use the xrefresh utility to redraw the entire display. To issue the command, merely type **xrefresh**. You can specify a particular host, server, and display as follows:

xrefresh *host:display:screen*

For information on the **host:display:screen** option, see Chapter 2.

4.6. Xset

Use the xset utility to set display preferences.

4.6.1. Command Format

Invoke *xset* with a command of the form:

xset b *volume pitch duration* **c** *volume* **fp** *path* **led** # **m** *acc thresh* **p** *tableno color* **r s** *time (no)blank host:display:screen*

The following table lists the options and their usage.

	OPTION	OPTION NAME	USAGE	DEFAULT
†	**b** *vol pitch duration*	bell	volume is % of maximum (0-100) pitch is in Hertz duration is in ms **b 0** turns bell off	0 400 100
†	**c** *vol*	key click	volume is % of maximum (0-100) **c 0** turns click off	0
	fp *path*	font path	This is the path used to locate fonts. The different directories should be separated by a comma. **fp default** sets the path to its default setting.	/usr/lib/X11/fonts
	*(-)***led** #	led on/off	# is the keyboard led number, where: • 1 is NUM LOCK led • 2 is CAPS LOCK led • 3 is SCROLL LOCK led **xset led** turns all three leds on **xset -led** turns all three leds off	 off off off
	m *acc thresh*	mouse	The cursor moves *acc* times as fast as the mouse. *thresh* is the number of pixels the mouse must move before X moves the cursor on the display. **m** sets the default *acc* may be entered without *thresh*	1 --
	p *tableno color*	pixel value	this changes the color at the specified *tableno* in the rgb data base to the specified *color* (see Appendix A)	none
	q	query	displays current xset settings	--
†	**r**	autorepeat	toggles autorepeat for keyboard keys	on
	s *time (no)blank*	display saver	*time* in seconds determines how long the display will sit with no input before display saver is turned on	600
			noblank unmaps the X application windows but leaves the root window displayed when display saver is on	noblank
			blank unmaps both the X application windows and the root window when display saver is on	noblank

† Where **x** is the option letter:

−x sets the option off

x with no flag resets the option to its default value(s)

x on turns the option on

x off turns the option off

4.7. Xsetroot

Xsetroot customizes the root window. To change the appearance of the root window permanently, place the xsetroot command in either:

- the .login file if X continuously runs on the display, or
- in the shell script that invokes X, if X is started by a UNIX command

4.7.1. Command Format

Use a command of the following format to customize the root window:

xsetroot -bg ***color*** **-bitmap** ***filename*** **-cursor** ***cursorfile maskfile*** **-def -fg** ***color*** **-name** ***string*** **-rv** ***host:display:screen***

-bitmap *filename*
The specified *filename*, a bitmap file, is tiled over the entire root window as a background. (For information on creating bitmaps, see Chapter 7.)

You can replace the **-bitmap** option with one of the following three options. Each is described later in this section, and defines a root window display style.

- **-gray** (or -grey)
- **-mod** *x y*
- **-solid** *color*

-cursor ***cursorfile maskfile***
This option uses the bitmap found in the *cursorfile* as the cursor in the root window. For information on creating bitmaps, see Chapter 7.

-def Use this option to return one or more options to their default setting. For example, use the command:

xsetroot -def

to set all options for this command to their default. (Note that you cannot use this option to reset an option you have set elsewhere on the same command line.)

-gray (or -grey)
This option displays the root window as a gray display composed of black and white pixels. This is not the same as specifying gray as a solid color with the **-solid** option.

-mod *x y*
This option paints the root window in a plaid pattern determined by the *x* and *y* entries (where *x* and *y* represent the distance between lines in a 16x16 bitmap). The *x* lines are vertical lines drawn in the background color (white for monochromes). The *y* lines are horizontal lines drawn in the foreground color (black for monochrome). This bitmap is tiled over the entire root window.

-name *string*
Use this option to set the name of the root window to *string*. The xwininfo command uses this name. There is no default value.

-rv This option reverses the foreground and background colors of the root window. You can use it with other options, such as **-mod** and **-bitmap**.

-solid *color*
This option sets the root window to *color*. You can use any of the names or numbers described in Appendix A for *color*.

See the *xsetroot* man page for more information on this command.

4.8. Xwd

The xwd command dumps a window image into a file. The window image can be redisplayed using the xwud command. It cannot be redisplayed using the bitmap editor.

4.8.1. Command Format

Use a command of the following format to dump a window image to a file:

xwd -nobdrs -out *filename* -xy *host:server.display*

The *xwd* options are described below.

-nobdrs
: This option dumps the window without its border.

-out *filename*
: This names the file into which the window is dumped. If the **-out** option is not specified, the window will be dumped to standard output. Dumping to standard output permits piping the contents of the window into a program (perhaps a print dump program). If xwd standard output is not directed to a program, then the contents of the window will display in dump format in the shell where the command was typed.

-xy
: On color displays, this option dumps the window in black and white (XY format). This option has no effect when specified for a monochrome display.

host:server.display
: This option specifies on which workstation and/or display the target window resides. See Chapter 2 for more information.

For more information about this command, see the *xwd* man page.

4.8.2. Dumping a Window

The procedure for dumping a window is as follows:

(1) Type the **xwd** command and options. The cursor changes to a target shape.

(2) Move the target cursor into the window to be dumped (the "target window").

(3) Click any mouse button. The bell sounds once at the beginning of the dump and twice at the end.

4.9. Xwininfo

The *xwininfo* command displays system information on the specified window. By default, the following information is displayed:

Tree: includes the IDs and names of the root, parent, and child windows associated with the selected window.

Events: lists the events for which the selected window is currently waiting.

Window Manager Hints: provides hints about how the window manager interacts with the selected window.

4.9.1. Command Format

Type a command of the following format to view system information for a specific window:

xwininfo -bits -id # -int -size -stats *host:server.display*

The rest of this section describes options of the xwininfo command.

-bits

Information on the window's raw bits is displayed when this option is specified. This information includes:

- bit gravity
- window gravity
- backing-store hint
- backing planes to be preserved
- backing pixel
- save under availability

-id # Use this option to select a window by typing its ID number rather than clicking the mouse in the selected window. This option is handy when the target window does not appear on the display, or when mouse clicks might interfere with the normal operation of an application.

One of these three options may be used in place of the **-id** option.

- **-font** *fontname*
- **-root**
- **-name** *windowname*

-int

With this option, you request that window IDs be displayed as integers rather than as hexadecimal numbers (the default).

-name *windowname*

Use this option to select a window by name rather than mouse click. For example, the command:

xwininfo -name xterm

displays information for an xterm window (provided it has not had its name altered by the **-n** option on the xterm command line).

Specifying the window by name rather than ID avoids confusion when two or more of the same type of window exist on the display at the same time.

-size

Use this option to request normal and zoom sizing hints for the selected window. Sizing hints include:

user-supplied location (offsets)
user-supplied size (=wxh)
program-supplied minimum size
program-supplied x resize increment
program-supplied y resize increment
program-supplied minimum aspect ratio
program-supplied maximum aspect ratio

-stats

This option provides statistics about the current state of the window. Statistics include:

upper left x pixel location
upper left y pixel location
window width
window height
window depth (refers to color)

border width
window class
window map state

4.9.2. Displaying X Window Information

To display X window information, proceed as follows:

(1) Type the **xwininfo** command and any desired options in the xterm window or other shell interface.

(2) If **-name**, **-root**, or **-id #** were not specified, the cursor assumes the target shape and the following prompt appears:

```
xwininfo = = > Please select the window you wish
         = = > information on by clicking the
         = = > mouse in that window.
```

(3) Click the mouse button in the desired window as prompted.

(4) The information appears in the window from which the command was entered. For more information on this command, see the *xwininfo* man page.

4.10. Xwud

This command "undumps" a window from a file created by *xwd*. The window image appears on the display at the exact pixel location from which it was originally dumped.

Windows that were dumped in color format (Z format) must be undumped on a color display.

4.10.1. Command Format

To undump a window image, type a command of the following format:

xwud -inverse -in *filename host:display:screen*

These options are described below.

-inverse
: This option undumps the image in reverse video (for monochrome displays only). This option is supplied because the display is "write white" (white = 1) and printers are generally "write black" (black = 1).

-in *filename*
: This option specifies the window dump file that will appear on the display. The default is to display standard input.

host:server.display
: This option specifies the host and/or display on which the window will appear. The image may appear a display other than the one from which it was dumped. It will appear in the corresponding pixel location from which it was originally dumped.

For more information on this command, see the *xwud* man page.

5. Customizing X

This chapter describes how to change the X default window characteristics, and how to configure X for particular environments.

5.1. Changing X Window Characteristics

X applications employ windows as part of the user interface. To change one or more characteristics of these windows, you change the .Xdefaults file. Each line in the file sets the default for one window characteristic and is of the form:

***Xapplication*.keyword:value**

where:

Xapplication is the name of the X application that presents the window

keyword is the name of the characteristic

value is the setting for that characteristic

Upper- and lowercase distinctions and extra spaces are ignored in these entries.

If you omit the **Xapplication** portion of the command, the new setting will affect all windows and/or menus. Such global defaults must precede all window-specific defaults in the .Xdefaults file. A window-specific setting will override a global setting if the window-specific setting appears after the global one in the file. The .Xdefaults file must reside in the home directory.

5.1.1. Window Keywords

Keywords are equivalent to X command line options. Setting the default for the keyword in the .Xdefaults file eliminates the need to specify the option on the command line each time you invoke the X application. If you specify a command line option when you invoke X from the UNIX shell, the option will override the keyword setting in the .Xdefaults file.

The following table lists all keywords, grouped by X application. The first set of keywords applies to all X applications. Use these keywords to make global settings. Use the remaining groups of keywords to set the appearance of windows belonging to particular X applications.

Note: Keywords must be capitalized as shown in the table,

KEYWORD	SETS:	RELATED OPTION
all windows:		
Background	background color	-bg
BodyFont	default font	-fn
Border	border color	-bd
BorderWidth	border width	-bw
Foreground	foreground color	-fg
ReverseVideo	foreground and background reversed	-r
bitmap:		
Highlight	hot spot color, also temporarily indicates move, copy, set, and invert areas. Inverted video is the default.	--
Mouse	mouse color	--
xclock:		
Highlight	color of the hands	-hl
InternalBorder	space between text and border (padding)	-padding
Mode	digital or analog	-digital or -analog
Update	update interval	-update
xfd:		
IconName	icon name	-in
Title	window title	-tl
xload:		
Highlight	color of hostname and scale lines	-hl
Scale	minimum scale	-scale
Update	update interval	-update
xterm:		
BoldFont	default bold font	-fb
C132	80 < = > 132 column switching capability	-132
Curses	curses fix	-cu
Cursor	text cursor color	-cr
CursorShape	cursor to arrow or I beam	--
Geometry	window size and position	= wxh ± xoff ± yoff
IconStartup	window or icon on creation	-i
InternalBorder	space between text and border (padding)	-b
Jump Scroll	jump scroll	-j
LogFile	log file name or pipe command	-lf
Logging	logging on/off	-ls
LogInhibit	inhibit logging	--
LoginShell	xterm to come up running login shell	-l
MarginBell	margin bell	-mb
Mouse	mouse color	-ms
MultiScroll	synchronous scroll	-s
NMarginBell	right margin	-nb
ReverseWrap	reverse wraparound mode	-rw
SaveLines	saved lines when scrollbar on	-sl
ScrollBar	scrollbar	-sb
ScrollInput	reposition on input with scrollbar	-si

KEYWORD	SETS:	RELATED OPTION
xterm (continued):		
SignalInhibit	inhibit signals from xterm menu	--
StatusLine	status line displayed on startup	--
StatusNormal	status line in normal video	--
VisualBell	visual rather than audio bell	-vb

5.1.2. Sample .Xdefaults File

The following is a sample .Xdefaults file. Note that the first three entries affect all windows. The remaining entries affect only the window type specified.

The "xterm.background" overrides the global ".background" entry. Generally, the last entered value, whether via a keyword in the appearance.

```
.background:          steel blue
.foreground:          goldenrod
.borderwidth:         5
bitmap.highlight:     firebrick
xclock.Highlight:     pink
xclock.update:        1
xclock.mode:          analog
xload.scale:          2
xterm.background:     magenta
xterm.bodyfont:       fg-13
xterm.cursorshape:    arrow
xterm.loginshell:     on
xterm.reversewrap:    on
cterm.scrollbar:      on
xterm.savedlines:     100
```

5.1.3. Activating X Window Defaults

To activate the X window characteristics in the .Xdefaults file for the root window, you use the *xrdb* command:

xrdb ***host:server:display*** **.Xdefaults**

There are three ways to invoke the command:

- Type it on an xterm command line
- Include it in the .login file (if X runs continuously)
- Include it in the startup shell script (if X is initiated by a UNIX shell command)

For more information, see the *xrdb* man page.

5.2. Configuring X

You can configure X to run continuously on a display, or to be invoked by command (either in a UNIX shell script or simply from the command line). This section explains how to configure X for each of these environments.

5.2.1. The xinit and Xibm Commands

Both startup configurations use the *xinit* and *Xibm* commands. The *xinit* command initializes *Xibm* and then initializes a specified X application. When the specified X

application terminates, then the *Xibm* server also terminates. *Xibm* actually starts the server. It is embedded in the *xinit* command line:

xinit ***Xapplication options*** **Xibm** ***options*** **&**

5.2.2. For X To Run Continuously

To configure X to run continuously on a display, you must change the /etc/ttys file and rename files in the /dev directory. Further, to cause an X application to start automatically for a user, you must modify that user's .login file. This section explains how to complete both these tasks.

5.2.2.1. Change /etc/ttys and /dev

Perform the following steps:

(1) Become the super user by typing the *su* command and password.

(2) Add the following line to the /etc/ttys file:

ttyv0 "/usr/bin/X11/xterm -L ***options*** **host:display";xterm on window = "/usr/bin/X11/Xibm** ***host:display options*** **-screen - screen - screen"**

Specify the **-L** option for the xterm window to set X for continuous display. This option causes X to present a window on which the user logs into the UNIX shell. Use the **-ls** option to request that the .login file be sourced automatically.

You may name more than one display by using **-display** flags. List the displays left to right on the command line as they rest left to right before the user. This will preserve an orderly left-to-right movement of the cursor between displays. The valid values for display names are:

apa16 for the IBM 6155 Extended Monochrome Graphics Display
aed for the IBM Academic Information Systems Experimental Display
mpel for the IBM 5081 Display with MegaPel adapter
ega for the IBM 5154 Enhanced Color Display with adapter
8514 for the IBM PS/2 Color Graphics Display Adapter 8514/A
vga for the IBM Video Graphics Array (VGA) Display

(3) Place a pound sign (#) at the beginning of the line that currently defines the display on which X will run. This "comments out" the line.

(4) In the /dev directory, type the following commands:

mv ttypf ttyv0 < Enter >
mv ptypf ptyv0 < Enter >

(5) Type the following:

kill -HUP 1 < Enter >

This reinitiates the /etc/ttys file, so the new display definition is read. X will come up in an xterm window on the first screen named in the ttyv0 command line. Your .login file will be sourced in this window. You can invoke additional xterm windows for other displays from the command line of your xterm window, or from a command in your .login file. The displays are numbered from 0 in the order their names appear in the command.

5.2.2.2. Change the User's .login File

Add the following lines at the end of a user's .login file to start an X application automatically on that user's display(s):

```
if ('tty' = = /dev/ttyv0) then
Xapplication options &
Xapplication options &
        .
        .
        .
endif
```

5.2.3. For X Invoked by Command

You can imbed the *xinit* command in a shell script to invoke Xibm from the UNIX shell. You can include the name of an X application that will be invoked as X is invoked. *Xterm* is usually the best choice, because exiting Xibm requires that you exit *xterm* as well, and *xterm* is useful during an entire work session.

Instead of the name of a single X application, you can substitute the name of a shell script that lists several initial applications, such as *xclock*, *xload*, and *xterm*.

You can also initiate X by simply typing **xinit -- Xibm** on the UNIX shell command line.

5.2.4. Sample Shell Script

A sample shell script is provided on the X/BE2 Installation diskette, in a file named *Demouwm/bin/X*. A sample start shell script is also provided, in a file named *Demouwm/bin/start*.

5.2.5. The Xibm Command

Use the *Xibm* command to start the X server. (The X server appears as the root window.) To start X applications as the server is started, use the *xinit* command. The following table lists the options for the Xibm command. See the *Xibm* man page for more information.

OPTION	OPTION NAME	USAGE
a #	mouse acceleration	# is a scale factor relating mouse movement to cursor movement.
c #	key click volume	# is the volume from a range of 0 through 8. A -c entry turns the key click off.
-d *display*	displays	each display on which this server is to run should be listed; see Sections 3.3.1 and 2.1.2 for more information
-f #	bell volume	# is the volume from a range of 0 through 8.
-fc *file*	cursor	*file* is a cursor bitmap file. This cursor will replace the default X cursor. T}
		-fn *font* + window font + T{ *font* may be replaced by any of the fonts listed in Appendix B, or any font along the path set by the **-fp** option
-fp *path*	font path	*path* sets the font path, directories are separated by a comma.
host:server	host and server number	host designates the workstation; server is a number that arbitrarily identifies the server in distinction from other servers on that workstation
-s #	screen saver timeout	sets the screen saver timeout in minutes
-t #	mouse threshold	# is the number of pixels the mouse must move before the cursor moves on the screen

6. Customizing Uwm

X includes a programmable window manager named *uwm*. You can customize *uwm* to your users' needs and preferences. This chapter describes the files and procedures needed to customize *uwm*.

_" . ds |n Starting Uwm"

6.1.

When you enter the command:

uwm &

the window manager is first configured using its internal default settings. Then it uses a search path to locate and process two startup files (in the order listed):

/usr/new/lib/X/uwm/system.uwmrc

$HOME/.uwmrc

If system.uwmrc exists, *uwm* adds these settings to its default settings. Any system.uwmrc file settings in conflict with the default settings override the default settings. (The latest setting always overrides previous settings.)

If $HOME/.uwmrc exists, *uwm* adds these settings to the combined default and system.uwmrc settings. Overrides occur as described above. This completes the process of configuring *uwm* for operation.

System.uwmrc file is a startup file that applies to all machines on the network. (The system administrator sets up this file.) $HOME/.uwmrc file is specific to a workstation. The default $HOME/.uwmrc file contains the commands that configure the default window manager.

To specify another file as the *uwm* startup file, modify the *uwm* command as follows:

uwm *-filename* &

where *filename* is a startup file you create. Specifying this file eliminates searching and reading *both* the system/.uwmrc and the $HOME/.uwmrc files. *Only* the settings in this file are added to *uwm* internal defaults.

6.2. Startup File Format

Startup files contain three parts which must appear in the order listed:

Global variables set characteristics for general *uwm* functions, such as the fonts used for icon names and menus.

Keybindings link combinations of keyboard and mouse actions to window manager functions. For example, pressing the right mouse button may present a menu.

Menu definitions list each menu selection and the command that executes it.

The rest of this chapter describes the elements of a startup file. However, most of the detail is in the *uwm* man page. Be sure you have access to it before starting to write your own startup file.

6.3. Global Variables

Global variables must appear first in the startup file. To ensure that only current startup file values are used, place the following three variables at the beginning of the file:

resetbindings

resetmenus

resetvariables

See the *uwm* man page for the rest of the global variables. They can appear in any order as long as they appear at the beginning of the file. If a variable does not appear in the file, it takes its value from the last file read. If it was not set in a previous file, the variable is assigned the *uwm* default setting.

6.4. Keybindings

Keybindings appear after global variables, and take the form:

function **= *control keys*:*context*:mouse actions:*"menu name"***

The *control keys*, *context*, and *"menu name"* entries are optional. Even though you can omit a *context* or *control keys* entry, you must include the colons and/or equal sign that precede and follow the entry; the command line must contain one equals sign and two colons. If you specify a *"menu name"* you must add the third colon.

Spaces may appear between the equals sign, colons, and an entry. You can use spaces to make the entries easier to read and edit. You may want to preface the keybindings with a comment line to serve as a heading, thus:

#FUNCTION=	KEYS	:CONTEXT:	MOUSE EVENTS	:"MENU NAME"
f.resize=	m	:w|i:	delta middle	

6.4.1. function

See the *uwm* man page for a description of the available functions.

6.4.2. control keys

Control keys include the following, which may be used alone or in combinations of two:

control or **c** (the Ctrl key on IBM keyboards) **meta** or **m** (the Alt key on IBM keyboards) **lock** or **l** (the Caps Lock key on IBM keyboards) **shift** or **s** (the Shift key on IBM keyboards)

NOTE: Although several keys on the IBM keyboard bear different labels, the words or letters in bold above *must* **be used in the file.**

To designate a a combination of two control key, separate them by a vertical bar (|). For example, the command:

f.move= m|s : window|icon: delta right

specifies that a window will be moved when the meta (Alt) and shift keys are pressed simultaneously.

If no control key is specified, mouse events invoke the named function. For example, the command:

f.move= : window|icon: delta right

specifies that a window will be moved when the right mouse button is pressed and the mouse is moved.

When defining keybindings, remember there are keybindings already defined by each X application. *Uwm* keybindings override X application keybindings, potentially crippling certain application features. For example, the keybinding example above, which specifies a delta right mouse move, will disable the cut capability of the right mouse button in the xterm window.

6.4.3. context

Context refers to the region where the cursor is located when keyboard and mouse actions occur. The four contexts are:

icon or i

window or w

root or r

any window (represented by a null entry)

You can combine any two of these contexts by using the vertical bar (|).

For example, the command:

f.iconify = meta :w|i: left down

is a toggle to change a window to an icon and back again.

6.4.4. mouse actions

The mouse buttons are identified as:

left or l

right or r

middle or m (both mouse buttons pressed simultaneously)

Each mouse button can be in one of three states:

down, when the button is pressed

up, when the button is released

delta, when the mouse has moved more than *delta* pixels (where the number of pixels is set by the **delta** variable)

For example, the command:

f.iconify = meta :w|i: l down

is a window/icon toggle activated when the left mouse button and the ALT key are pressed.

The range of mouse actions is:

right down	left down	middle down
right up	left up	middle up
delta right	delta left	delta middle

Note the mouse button name follows the word *delta*, but precedes the words *up* and *down*.

6.4.5. *menu name*

Using the f.menu function requires a *"menu name"* entry, which must match exactly the name as it appears on the top of the displayed menu. The *"menu name"* is the last entry on the keybinding command line, and is preceded by a colon.

For example, the command

f.menu = :: right down: "Main Menu"

causes a menu named "Main Menu" to appear when the right mouse button is pressed.

6.4.6. Slip-off Keybindings

Because a different function can be tied to the up, down, and delta states for each mouse button, you can tie related sequential functions to the mouse action sequence down, delta, up.

For example, f.iconify may be tied to right down, f.move to delta right, and f.raise to right up. The result is the window/icon changes its form on right down, the window/icon moves on delta right, and the window/icon appears at the top of the stack on right up.

6.4.7. Slip-off Menus

You can define menus so that simply moving the mouse out of one menu will bring up the next menu in that series. These are known as slip-off menus.

All menus in a slip-off menu series are bound to exactly the same control keys, context, and mouse actions.

The f.menu entries are listed one after the other in the keybindings section of the startup file. The first menu named is the first that appears when the associated key and mouse actions occur. The next listed menu appears when the cursor slides out of the first menu. The third listed menu appears when the cursor slides out of the second menu, and so on. When the last menu in the series appears, *uwm* does not return to the first menu. The user must again use the key and mouse actions to restart the series.

For example, the following keybindings define a slip-off menu series.

#FUNCTION=	KEYS	:CONTEXT:	MOUSE EVENTS	:"MENU NAME"
f.menu =		::	right down	:"Main Menu"
f.menu =		::	right down	:"Manage a Window"
f.menu =		::	right down	:"Create a Window"
f.menu =		::	right down	:"Manage the Display"

6.5. Menu Definition

The menu definition part of the startup file must contain one menu definition for each f.menu function that was listed in the keybindings part of the startup file. Menu definitions use the following format:

menu = "menu name" *(fghd:bghd:fghl:bghl)* {
"selection name" : *(fg:bg):* "action"
"selection name" : *(fg:bg):* "action"
}

The *"menu name"* is the same as the one specified in the keybinding line. The *selection name* is a menu selection that can be chosen by mouse click. The *action* is the process that this selection triggers.

The selection names and actions must be enclosed in double quotes if they contain quotes, special characters, parentheses, tabs, or blanks.

The *fg, bg, fghl,* and *bghl* entries set colors (only on color displays). If the display is monochrome, or if the default settings are acceptable, these entry fields can be omitted. A menu cannot scroll, but can contain as many selections as will fit on the display.

6.5.1. Menu Actions

There are three types of menu actions:

Window manager functions are defined in the uwm man page.

Shell commands must begin with exclamation point (!) and end with an ampersand (&). If the command includes spaces or special characters, enclose the command in double quotes. For example, the action:

```
!"xterm -rv &"
```

creates a reverse-video xterm window when its corresponding menu selection is chosen.

A text string is placed in the server's cut buffer when the string's corresponding menu selection is chosen. Once in the buffer, the string may be pasted into xterm or any other window that provides cut and paste facilities. This is handy if a particularly long text string is frequently used. For information on pasting the string into an individual window, see the paste instructions for that window.

When the menu action is a text string, it muse be preceded by one of two special characters:

The caret (^) precedes the entire string if it contains any newline characters, such as carriage return or line feed:

```
"^cd /usr/doc/ibmdoc/smm
ls -al"
```

The vertical bar (|) precedes the entire string if does not contain a newline character: "|tbl x00 x01 x02 | ptroff -me"

6.5.2. Adding Color to Menus

The color designations in the menu definition correspond to the various menu regions as follows: • **fghd**: foreground color for menu header • **bghd**: background color for menu header • **fghl**: foreground color for highlighted selection • **bghl**: background color for highlighted selection • **fg**: foreground color for rest of menu • **bg**: background color for rest of menu

Appendix A lists available colors.

Colors specifications in a file used to configure a monochrome display are ignored; the display uses the default black/white settings.

On a color display, the colors default to the foreground and background colors of the root window if any of the following is true:

- The number of color map entries has been exceeded.
- Either a foreground or a background color does not exist in the rgb data base; this pair of colors uses the default.

- Either a foreground or background color is omitted; this pair of colors will use the default.
- The number of colors specified exceeds the **maxcolors** variable (see *uwm* man page).
- No colors are specified.

6.6. Sample Uwm File

A sample /fluwm file is provided on the X/BE2 Installation diskette, in a file named *Demouwm/src/.uwmrc*.

7. The Bitmap Editor

X includes an editor to facilitate creating and editing bitmap files. A bitmap is a rectangular array of black and white pixels (1 and 0 bits) that form graphic displays that used as cursors, icons, and tiles in the root window.

The bitmap window presents a magnified view of the rectangular array. A grid divides the rectangle into boxes, each box representing one pixel. You turn individual pixels on and off using the mouse cursor and/or command buttons that appear to the right of the grid. You also use the command buttons to access other functions, such as drawing lines and circles, and operating on a specified area within the rectangle.

Below the *Quit* button in the window, you see a size representation of the bitmap. Beneath this, there is a reverse video version of the bitmap drawn to scale.

When the editor writes a file, it also writes a program fragment. You can include this fragment in C programs or use it with X commands to simplify the process of defining cursor and icon shapes and sizes. Use *#include* to include the fragment in a C program. Use the fragment with such X command as *X*, *xsetroot*, and *xterm*.

7.1. Masks for Cursor Bitmaps

Whereas most bitmap files are rectangular, a cursor image occupies only a portion of its rectangular area. You can set the non-cursor portion of the bitmap to "transparent" so that the cursor is not just a square that contains a shape. To do so, create two bitmap files, one for the cursor and one for an overlay or "mask."

When you use this approach, keep the following in mind:

- All bits set to 0 in the mask are transparent, no matter how the overlapping bits in the cursor file are set. That is, the mask bit value overrides the cursor bit value.
- When a mask bit and its corresponding cursor bit are both set to 1, then the bit appears in the foreground color.

(bu When a mask bit is set to 1 and its corresponding cursor bit is set to 0, then the bit appears in the background color.

7.2. The *Bitmap* Command

To invoke the bitmap editor, use the following command:

bitmap filename dimensions =wxh ± xoff ± yoff host:server.display

The *filename* and *dimensions* options are described below. See the *bitmap* man page for a complete discussion of this command.

7.2.1. filename

You must include a file name as the first parameter of the *bitmap* command. Otherwise, an error message appears. If the file specified doesn't exist, a new file is created. Use normal UNIX file naming conventions to name the file.

An existing file must be in bitmap format. (Remember that a file dumped by the xwd command cannot be edited by bitmap.) For a description of bitmap file format, see the *bitmap* man page.

7.2.2. dimensions

The **dimensions** are the width and height of a new bitmap in pixels. The default is 16x16. You cannot use this option to change the dimensions of an existing bitmap file.

7.2.3. Error Messages

If the system detects an error in the *bitmap* command you enter, one of the following messages appears:

ERROR MESSAGE	OCCURS WHEN
Bitmap displays these messages and then aborts:	
could not connect to server on *host:server.display*	• incorrect DISPLAY variable • specified host is down • home workstation is not in/etc/xhosts file on specified host • host is not running X • host is refusing connections
could not open file *filename* **for reading** -- *message*	specified file exists but could not be read for the reason listed in *message*
dimensions must be positive	negative dimensions were entered
file *filename* **does not have a valid width dimension**	the input file does not have the correct format
file *filename* **does not have a valid height dimension**	the input file does not have the correct format
file *filename* **has an invalid** *n***th array element**	the input file does not have the correct format
invalid dimension *'string'*	the dimensions were incorrectly entered or were out of range
Bitmap displays these messages in xterm after creating a window:	
Unrecognized variable *name* **in file** *filename*	*filename* contains a variable ending in something other than *_x_hot, _y_hot, _width, or _height*
XError:*message*	there is a protocol error, i.e.: • the X server is malfunctioning • the X library is in error • the X server and library are incompatible • the X connection has been broken
XIOError	same as conditions for **XError**

7.3. Using the Editor

You use the command buttons and mouse to draw a bitmap. As you draw the bitmap, its actual size representation appears in normal and reverse video in areas to the right of the grid.

7.3.1. Color Conventions

In bitmap, when you "set" a pixel (set it to 1), it appears in the foreground color. When you "clear" a pixel (set it to 0), it appears in the background color. Whenever

you change a pixel setting, the change appears in the normal and reverse video areas. (You specify the foreground and background colors in the .Xdefaults file described in Chapter 4.)

7.3.2. Command Buttons

Be sure to use the command buttons on a normal (not reverse video) grid. Areas drawn on a reverse video grid with the command buttons may not appear as you expect.

The command buttons and their functions are as follows:

Clear All	Change all pixels to 0, the background color
Set All	Change all pixels to 1, the foreground color
Invert All	Change all set pixels to clear, all clear pixels to set
Clear Area	Change all pixels in the defined area to 0
Set Area	Change all pixels in the defined area to 1
Invert Area	In the defined area, change set pixels to clear and vice versa
Copy Area	Copy the defined area to another location
Move Area	Move the defined area to a new location
Overlay Area	Combine the defined area with another
Line	Draw a line between two points
Circle	Draw a circle with the specified center and radius
Filled Circle	Draw a circle filled with the foreground color
Set HotSpot	Specify the pixel that is the exact pointer of the cursor
Clear HotSpot	Clear any previously-set hot spot
Write Output	Write this bitmap to the file named in the *bitmap* command
Quit	End this editor session

7.3.3. Selecting a Command Button

To select a command, move the mouse cursor into the command button box and click either mouse button. The command button box becomes highlighted.

7.3.4. Command Button Operation

Several command buttons execute automatically when you select them, because they require no further input. When the operation is complete, the button reverts to its normal color. The automatic command buttons are as follows:

Clear All
Set All
Invert All
Clear HotSpot
Write Output
Quit (if preceded by a Write Output)

Other command buttons cause the following changes in the bitmap window:

(1) After you select the command button, the appearance of the cursor changes.

(2) The command button remains highlighted while you provide input (mouse movement and clicks). The cursor may change appearance when you click the mouse, especially if the command requires more than one mouse clicks.

(3) After you have provided all necessary input, the cursor reverts to its normal shape and the command button reverts to its normal color.

7.4. Drawing a Bitmap

Drawing a bitmap is nothing more than setting, clearing, and inverting pixels. You can do so by working a pixel at a time, or by working with a defined area.

7.4.1. Setting, Clearing, and Inverting Individual Pixels

You set, clear, or invert an individual pixel as follows:

(1) Move the target cursor into one of the grid boxes.

(2) Click the the appropriate mouse button(s):

To set a pixel, click the left mouse button.

To clear a pixel, click the right mouse button.

To invert a pixel, click both mouse buttons.

7.4.2. Drawing Lines, Curves, and Circles

You can use the mouse to draw freehand lines, curves, and circles in either the background or foreground color. You can use the command buttons to draw these same shapes, but only in the foreground color.

7.4.2.1. Freehand Drawing

You can draw any line or curve with the mouse. Just hold down the appropriate mouse button to set, clear, or invert the pixels as you move the mouse. Move the mouse slowly to ensure that all pixels in its path are set or cleared correctly.

7.4.2.2. Line Command Button

You can use this command button to draw a line in the foreground color using only three mouse clicks.

(1) Move the cursor into the *Line* command button and click any mouse button.

(2) Move the cursor to the grid box at which the line is to begin, and click any mouse button. An X appears in that box to show the beginning of the line.

(3) Move the mouse cursor to the grid box at which the line is to end, and click any mouse button. The boxes between the starting point and end point of the line are set to the foreground color.

7.4.2.3. Circle Command Button

You can use this command button to set a circle of pixels in the foreground color using only three mouse clicks.

(1) Move the cursor into the *Circle* command button and click any mouse button.

(2) Move the cursor to the box that represents the center of the circle, and click any mouse button. An X appears to mark the circle's center.

(3) Move the cursor to a box at the outer edge of the circle, and click any mouse button. The distance from the centerpoint to this box is the radius of the circle. The circle will appear.

7.4.2.4. Filled Circle

The *Filled Circle* command button works exactly the same as the *Circle* command button. The only difference is that all pixels within the circle are set when the circle is drawn.

7.4.3. Area Operations

The following command buttons perform operations on an area of the bitmap:

Clear Area
Set Area
Invert Area
Copy Area
Move Area
Overlay Area

7.4.4. Clear Area, Set Area, Invert Area

These buttons have the same effect as the *Clear All*, *Set All*, and *Invert All* buttons, except the effect is limited to an area you specify with mouse clicks:

(1) Move the cursor to the correct command button and click any mouse button. An angled arrow that points to the upper left corner of the grid appears.

(2) Move this arrow to the box that represents the upper left corner of the area to be cleared, set, or inverted.

(3) Hold down any mouse button. The cursor changes to an angled arrow that points to the lower right corner of the grid.

(4) Still holding down the mouse button, move the cursor down and to the right until you reach the box that represents the lower right corner of the area. An X fills each box in the selected area. (If you move the cursor up and to the left, and then release the mouse button, you cancel the operation.)

(5) Release the mouse button to complete the clear, set, or invert operation.

7.4.5. Copy Area, Move Area, Overlay Area

These functions operate as follows:

Copy Area
leaves the pattern in the original area, and copies it to another area, destroying any existing pattern in the new area.

Move Area
removes the pattern from the original area, and places it in the new area, destroying any pattern in the new area.

Overlay Area
leaves the pattern in the original area, and superimposes that pattern in another area. If a pattern already existed in the new area, it is combined with the overlaying pattern.

Use any of these command buttons as follows:

(1) Move the cursor to the correct command button and click any mouse button. An angled arrow that points to the upper left corner of the grid appears.

(2) Move this arrow to the box that represents the upper left corner of the area to be cleared, set, or inverted.

(3) Hold down any mouse button. The cursor changes to an angled arrow that points to the lower right corner of the grid.

(4) Still holding down the mouse button, move the cursor down and to the right until you reach the box that represents the lower right corner of the area. An X fills each box in the selected area. (If you move the cursor up and to the left, and then release the mouse button, you cancel the operation.)

(5) Release the mouse button. The cursor changes back to the left-angled arrow.

(6) Move the cursor to the box that is the upper left corner of the new area.

(7) Press any mouse button. The selected pattern will be copied, moved or overlaid in the designated area.

7.4.6. The Hot Spot

The hot spot is the pixel within the bitmap that the X server perceives as "active." Usually this is a portion of the cursor. X does not keep track of entire bitmaps as they move on the display. X only tracks the hot spot.

For example, you may use the bitmap editor to draw an arrow that will be used as a cursor. You should set the hot spot to the pixel at the tip of the arrow. Whenever the tip of the arrow crosses a window boundary, X considers the cursor to be in the new window, even though much of its image on the display may be in the old window.

7.4.6.1. Set HotSpot, Clear HotSpot

Use the following instructions to set a hot spot. If a hot spot exists when you set a new hot spot, the old hot spot is erased. Only the new hot spot remains in effect.

(1) Click any mouse button in the *Set HotSpot* command button. The shape of the cursor changes.

(2) Move the cursor to the grid box designated for the hot spot. (this can be a set or a cleared pixel.)

(3) Click any mouse button. A diamond shape appears in the selected box, which is now the hot spot for this bitmap.

To clear a hot spot, click any mouse button in the *Clear HotSpot* command button. The hot spot disappears from the bitmap.

7.5. Saving the Bitmap

You can save the bitmap in the grid at any time by clicking any mouse button in the *Write Output* command button. The *Write Output* button flashes in the foreground color and the file is saved, using the **filename** specified on the command line.

If no path name is specified with the **filename** on the command line, the file is stored in the directory from which the original **bitmap** command was issued in the xterm window.

7.6. Exiting Bitmap: Quit

Exit the bitmap editor via the *Quit* command button, as follows:

(1) Click any mouse button in the *Quit* command button. If no changes were made to the bitmap since the last time the file was written or since the window was first opened, the bitmap window closes.

(2) If changes were made too the bitmap, a window with the words *Save changes before quitting?* appears in the upper left corner of the bitmap window.

(3) Click any mouse button in one of this box's command buttons. The selected button will have the following effect:

Yes writes the bitmap to a file, and quits.

No quits without writing the bitmap to a file.

Cancel
Cancels the quit command and returns to the editor.

7.7. File Format

For information on bitmap file format, see the *bitmap* man page.

Appendix A: X Colors

The following list specifies the colors available with X. The list also appears in /usr/lib/rgb.txt.

The hexadecimal numbers to the left of the color name represent the intensities of red, green, and blue respectively required to make the named color. You can specify a color in an X command by name or by hexadecimal number. The hexadecimal numbers must be in one of the following formats, where R = red, G = green, and B = blue:

#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB

To create your own color, enter a number combination on the command line (following the appropriate option flag). For example, the following command:

xterm -fg #567239042 &

invokes *xterm* with a foreground color equivalent to the specified hexadecimal mixture of red, green, and blue.

When using color names, type the names *exactly* as they are listed, paying particular attention to upper- and lowercase characters and spaces. If a color name includes spaces, enclose the name in double quotes. For example, either of the following commands:

xterm -fg "cadet blue" &
xterm -fg CadetBlue &

invokes *xterm* with Cadet Blue for the foreground color.

112 219 147	aquamarine
112 219 147	Aquamarine
50 204 153	medium aquamarine
50 204 153	MediumAquamarine
0 0 0	black
0 0 0	Black
0 0 255	blue
0 0 255	Blue
95 159 159	cadet blue
95 159 159	CadetBlue
66 66 111	cornflower blue
66 66 111	CornflowerBlue
107 35 142	dark slate blue
107 35 142	DarkSlateBlue
191 216 216	light blue
191 216 216	LightBlue
143 143 188	light steel blue
143 143 188	LightSteelBlue
50 50 204	medium blue
50 50 204	MediumBlue
127 0 255	medium slate blue
127 0 255	MediumSlateBlue
47 47 79	midnight blue
47 47 79	MidnightBlue
35 35 142	navy blue

```
35 35 142       NavyBlue
35 35 142       navy
35 35 142       Navy
50 153 204      sky blue
50 153 204      SkyBlue
0 127 255       slate blue
0 127 255       SlateBlue
35 107 142      steel blue
35 107 142      SteelBlue
255 127 0       coral
255 127 0       Coral
0 255 255       cyan
0 255 255       Cyan
142 35 35       firebrick
142 35 35       Firebrick
204 127 50      gold
204 127 50      Gold
219 219 112     goldenrod
219 219 112     Goldenrod
234 234 173     medium goldenrod
234 234 173     MediumGoldenrod
0 255 0         green
0 255 0         Green
47 79 47        dark green
47 79 47        DarkGreen
79 79 47        dark olive green
79 79 47        DarkOliveGreen
35 142 35       forest green
35 142 35       ForestGreen
50 204 50       lime green
50 204 50       LimeGreen
107 142 35      medium forest green
107 142 35      MediumForestGreen
66 111 66       medium sea green
66 111 66       MediumSeaGreen
127 255 0       medium spring green
127 255 0       MediumSpringGreen
143 188 143     pale green
143 188 143     PaleGreen
35 142 107      sea green
35 142 107      SeaGreen
0 255 127       spring green
0 255 127       SpringGreen
153 204 50      yellow green
153 204 50      YellowGreen
47 79 79        dark slate grey
47 79 79        DarkSlateGrey
47 79 79        dark slate gray
47 79 79        DarkSlateGray
84 84 84        dim grey
84 84 84        DimGrey
84 84 84        dim gray
84 84 84        DimGray
```

```
168 168 168     light grey
168 168 168     LightGrey
168 168 168     light gray
168 168 168     LightGray
159 159 95      khaki
159 159 95      Khaki
255 0 255       magenta
255 0 255       Magenta
142 35 107      maroon
142 35 107      Maroon
204 50 50       orange
204 50 50       Orange
219 112 219     orchid
219 112 219     Orchid
153 50 204      dark orchid
153 50 204      DarkOrchid
147 112 219     medium orchid
147 112 219     MediumOrchid
188 143 143     pink
188 143 143     Pink
234 173 234     plum
234 173 234     Plum
255 0 0         red
255 0 0         Red
79 47 47        indian red
79 47 47        IndianRed
219 112 147     medium violet red
219 112 147     MediumVioletRed
255 0 127       orange red
255 0 127       OrangeRed
204 50 153      violet red
204 50 153      VioletRed
111 66 66       salmon
111 66 66       Salmon
142 107 35      sienna
142 107 35      Sienna
219 147 112     tan
219 147 112     Tan
216 191 216     thistle
216 191 216     Thistle
173 234 234     turquoise
173 234 234     Turquoise
112 147 219     dark turquoise
112 147 219     DarkTurquoise
112 219 219     medium turquoise
112 219 219     ediumTurquoise
79 47 79        violet
79 47 79        Violet
159 95 159      blue violet
159 95 159      BlueViolet
216 216 191     wheat
216 216 191     Wheat
252 252 252     white
```

```
252 252 252     White
255 255 0       yellow
255 255 0       Yellow
147 219 112     green yellow
147 219 112     GreenYellow
```

lat-s30.snf
life1.snf
mailfont12.snf
met25.snf
micro.snf
mit.snf
oldera.snf
pe.12
pe.12.snf
plunk.snf
ree
rot-s16.snf
runlen.snf
s.30
s.30.snf
s.bold-italic.30
s.bold-italic.30.snf
s.bold.30
s.bold.30.snf
s.italic.30
s.italic.30.snf
script12.snf
script12b.snf
script12bi.snf
script12i.snf
shape10.snf
ss.30
ss.30.snf
ss.bold-italic.30
ss.bold-italic.30.snf
ss.bold.30
ss.bold.30.snf
ss.italic.30
ss.italic.30.snf
stan.snf
stempl.snf
sub.snf
subsub.snf
sup.snf
supsup.snf
swd-s30.snf
sym-s25.snf
sym-s53.snf
sym10.snf
sym12.snf
sym12b.snf
sym8.snf
table12.snf
tri10.snf
variable
variable.snf
vbee-36.snf
vctl-25.snf
vg-13.snf
vg-20.snf
vg-25.snf
vg-31.snf
vg-40.snf
vgb-25.snf
vgb-31.snf
vgbc-25.snf
vgh-25.snf
vgi-20.snf
vgi-25.snf
vgi-31.snf
vgl-40.snf
vgvb-31.snf
vmic-25.snf
vply-36.snf
vr-20.snf
vr-25.snf
vr-27.snf
vr-30.snf
vr-31.snf
vr-40.snf
vrb-25.snf
vrb-30.snf
vrb-31.snf
vrb-35.snf
vrb-37.snf
vri-25.snf
vri-30.snf
vri-31.snf
vri-40.snf
vsg-114.snf
vsgn-57.snf
vshd-40.snf
vxms-37.snf
vxms-43.snf
xif-s25.snf
zipicon12.snf
ziticon12.snf

91	=	[	144	=	M-^P
92	=	\	145	=	M-^Q
93	=	]	146	=	M-^R
94	=	^	147	=	M-^S
95	=	_	148	=	M-^T
96	=	`	149	=	M-^U
97	=	a	150	=	M-^V
98	=	b	151	=	M-^W
99	=	c	152	=	M-^X
100	=	d	153	=	M-^Y
101	=	e	154	=	M-^Z
102	=	f	155	=	M-^[
103	=	g	156	=	M-^\
104	=	h	157	=	M-^]
105	=	i	158	=	M-^^
106	=	j	159	=	M-^_
107	=	k	160	=	M-
108	=	l	161	=	M-!
109	=	m	162	=	M-"
110	=	n	163	=	M-#
111	=	o	164	=	M-$
112	=	p	165	=	M-%
113	=	q	166	=	M-&
114	=	r	167	=	M-'
115	=	s	168	=	M-(
116	=	t	169	=	M-)
117	=	u	170	=	M-*
118	=	v	171	=	M-+
119	=	w	172	=	M-,
120	=	x	173	=	M--
121	=	y	174	=	M-.
122	=	z	175	=	M-/
123	=	{	176	=	M-0
124	=	|	177	=	M-1
125	=	}	178	=	M-2
126	=	~	179	=	M-3
127	=	^?	180	=	M-4
128	=	M-^@	181	=	M-5
129	=	M-^A	182	=	M-6
130	=	M-^B	183	=	M-7
131	=	M-^C	184	=	M-8
132	=	M-^D	185	=	M-9
133	=	M-^E	186	=	M-:
134	=	M-^F	187	=	M-;
135	=	M-^G	188	=	M-<
136	=	M-^H	189	=	M-=
137	=	M-^I	190	=	M->
138	=	M-^J	191	=	M-?
139	=	M-^K	192	=	M-@
140	=	M-^L	193	=	M-A
141	=	M-^M	194	=	M-B
142	=	M-^N	195	=	M-C
143	=	M-^O	196	=	M-D

Appendix D: Xterm Escape Sequences

This appendix lists the DEC VT102 escape sequences, as well as special sequences used by such *xterm* features as the scrollbar. An xterm window can receive these sequences from a program or from the *echo* command:

echo "*escape sequence*" < Enter >

For example, the escape sequence that sets wraparound mode is:

Esc [? 7 h

This sequence can be echoed as follows. Note that typing the Escape key produces a ^[on the display.

echo "^[[?7h"

The escape sequences list begins on the next page.

C = || B || → **United States (USASCII)**

|| ESC || 7 || Save Cursor (DECSC)

|| ESC || 8 || Restore Cursor (DECRC)

|| ESC || = || Application Keypad (DECPAM)

|| ESC || > || Normal Keypad (DECPNM)

|| ESC || D || Index (IND)

|| ESC || E || Next Line (NEL)

|| ESC || H || Tab Set (HTS)

|| ESC || M || Reverse Index (RI)

|| ESC || N || Single Shift Select of G2 Character Set (SS2)

|| ESC || O || Single Shift Select of G3 Character Set (SS3)

|| ESC || T || P_s || LF || Change Window Title to P_s

|| ESC || [|| P_s || @ || Insert P_s (Blank) Character(s) (default = 1) (ICH)

|| ESC || [|| P_s || A || Cursor Up P_s Times (default = 1) (CUU)

|| ESC || [|| P_s || B || Cursor Down P_s Times (default = 1) (CUD)

|| ESC || [|| P_s || C || Cursor Forward P_s Times (default = 1) (CUF)

|| ESC || [|| P_s || D || Cursor Backward P_s Times (default = 1) (CUB)

|| ESC || [|| P_s || ; || P_s || H ||
Cursor Position [row;column] (default = [1,1]) (CUP)

|| ESC || [|| P_s || J || Erase in Display (ED)
P_s = || 0 || → Clear Below (default)
P_s = || 1 || → Clear Above
P_s = || 2 || → Clear All

|| ESC || [|| P_s || K || Erase in Line (EL)
P_s = || 0 || → Clear to Right (default)
P_s = || 1 || → Clear to Left
P_s = || 2 || → Clear All

|| ESC || [|| P_s || L || Insert P_s Line(s) (default = 1) (IL)

|| ESC || [|| P_s || M || Delete P_s Line(s) (default = 1) (DL)

|| ESC || [|| P_s || P || Delete P_s Character(s) (default = 1) (DCH)

|| ESC || [|| P_s || c || Device Atrributes (DA1)

|| ESC || [|| P_s || ; || P_s || f ||
Cursor Position [row;column] (default = [1,1]) (HVP)

|| ESC || [|| P_s || g || Tab Clear
P_s = || 0 || → Clear Current Column (default)
P_s = || 3 || → Clear All

|| ESC || [|| P_s || h || Mode Set (SET)
P_s = || 4 || → Insert Mode (IRM)
P_s = || 2 || || 0 || → Automatic Linefeed (LNM)

|| ESC || [|| P_s || l || Mode Reset (RST)
P_s = || 4 || → Insert Mode (IRM)

P_s = || 4 || 8 || → **Reverse Status Line**

|| ESC || [|| ? || P_s || l ||

DEC Private Mode Reset (DECRST)

P_s = || 1 || → Normal Cursor Keys (DECCKM)

P_s = || 3 || → 80 Column Mode (DECCOLM)

P_s = || 4 || → Jump (Fast) Scroll (DECSCLM)

P_s = || 5 || → Normal Video (DECSCNM)

P_s = || 6 || → Normal Cursor Mode (DECOM)

P_s = || 7 || → No Wraparound Mode (DECAWM)

P_s = || 8 || → No Auto-repeat Keys (DECARM)

P_s = || 9 || → Don't Send MIT Mouse Row & Column on Button Press

P_s = || 4 || 0 || → Disallow 80 ↔ 132 Mode

P_s = || 4 || 1 || → No *curses*(5) fix

P_s = || 4 || 4 || → Turn Off Margin Bell

P_s = || 4 || 5 || → No Reverse-wraparound Mode

P_s = || 4 || 6 || → Stop Logging

P_s = || 4 || 7 || → Use Normal Screen Buffer

P_s = || 4 || 8 || → Un-reverse Status Line

|| ESC || [|| ? || P_s || r ||

Restore DEC Private Mode

P_s = || 1 || → Normal/Application Cursor Keys (DECCKM)

P_s = || 3 || → 80/132 Column Mode (DECCOLM)

P_s = || 4 || → Jump (Fast)/Smooth (Slow) Scroll (DECSCLM)

P_s = || 5 || → Normal/Reverse Video (DECSCNM)

P_s = || 6 || → Normal/Origin Cursor Mode (DECOM)

P_s = || 7 || → No Wraparound/Wraparound Mode (DECAWM)

P_s = || 8 || → Auto-repeat/No Auto-repeat Keys (DECARM)

P_s = || 9 || → Don't Send/Send MIT Mouse Row & Column on Button Press

P_s = || 4 || 0 || → Disallow/Allow 80 ↔ 132 Mode

P_s = || 4 || 1 || → Off/On *curses*(5) fix

P_s = || 4 || 4 || → Turn Off/On Margin Bell

P_s = || 4 || 5 || → No Reverse-wraparound/Reverse-wraparound Mode

P_s = || 4 || 6 || → Stop/Start Logging

P_s = || 4 || 7 || → Use Normal/Alternate Screen Buffer

P_s = || 4 || 8 || → Un-reverse/Reverse Status Line

|| ESC || [|| ? || P_s || s ||

Save DEC Private Mode

P_s = || 1 || → Normal/Application Cursor Keys (DECCKM)

P_s = || 3 || → 80/132 Column Mode (DECCOLM)

P_s = || 4 || → Jump (Fast)/Smooth (Slow) Scroll (DECSCLM)