Volume 3. User Guide

Contents

1. Invoking A+

Invocation From the Shell; Environment From Within an Emacs Session

If you start an Emacs session by selecting **Emacs/A+** from the main menu, A+ can be invoked within it (with the resulting buffer in a-mode) by pressing the function key **F4**. When in Emacs, you can modify the effect of the **F4** key by entering **Control-c a**.

Concurrent Sessions

You can run several Emacs A+ sessions concurrently. You can run A+ in several concurrent Emacs sessions. Moreover, to run concurrent A+ sessions within a single Emacs session, after starting one A+ session, you can rename it by pressing **Escape x** and entering `M-x` **`rename-buffer`** *`newname`*, then perhaps change to another release using **Control-c a**, then start another session by pressing **F4**, then perhaps rename that session and start another one, and so on.

From a Shell Command Line

A+ can also be invoked from a shell command line, by
`a+` *`[options] [script [args]]`*, where the meaning of the parameters is as follows.

- *options:*
  `-d` *`display`*
  Set the environment variable DISPLAY to *`display`*.
  `-h`
  Set the heap size to 128 megabytes. CAUTION: Do *not* use this flag unless it specifically solves a memory related problem.
  `-m` *`memmap`*
  `memmap` is one of the memory mapping methods for atmp described in the "Work Area" section (Releases 2.42 and 4.09 on).
  `-q`
  Suppress the banner (Releases 2.42 and 4.09 on).
  `-w` *`wssize`*
  Set the initial workspace size to *`wssize`* (in megabytes).

- *script:*
  The name of the script to be loaded when A+ is invoked.

- *args:*
  The parameter list for the script, which is made available to the script by the special system variable `_argv`.
  The value of `_argv` is a vector of character strings.
  Following the Unix and C language conventions,
  `0⊃_argv` is the name of the script,
  `1⊃_argv` is the first parameter,
  `2⊃_argv` is the second, etc.

The PWD (environment name) environment variable is set on startup.

If you want to suppress all error reports (messages to stderr) for the session, you can put **`2>/dev/null`** at the end of the shell command line.

Examples

```
/usr/local/bin/a+
```

Invoke A+ using all of the defaults (the default workspace size is 1 MB, or 2^{20} bytes.)

```
/usr/local/bin/a+ -w 100
```

Invoke A+ with a 100MB virtual workspace.

```
/usr/local/bin/a+ -d workstation-name:0.0 graphit expertlevel
```

Invoke A+ with the environment variable DISPLAY set to *workstation-name:0.0*, and load the script named graphit with the parameter expertlevel. The value of `_argv` in the A+ session will be `("graphit";"expertlevel")`.

A+ uses the search rules described for `$load` to find the application script *graphit*. The rules are based on the environment variable APATH. The default setting of APATH is:

```
.:/usr/local/aplus-fsf-n.nn/lib
```

APATH is automatically set by A+ on startup and its value is derived from another automatically set A+ environment variable, ATREE. If either if these environment variables exist prior to starting A+ the existing values are used.

It is very important for these variables to be set correctly and it is recommended that A+ set them. If set incorrectly, the wrong versions of s and adap will be loaded.

If customization of APATH is required, it is recommended that it is done after A+ has started:

```
sys.setenv{"APATH=/look/here/first:",sys.readenv "APATH"}
```

Correct commands:

```
sys.system{"unset APATH ATREE;/usr/local/bin/a+"}
$(unset APATH ATREE;/usr/local/bin/a+)
```

Note: When an A+ process initiates a second A+ process, the second process inherits the value of ATREE and APATH from the first, even if the process is detached. To avoid this, do, e.g.,

```
sys.system{"unset ATREE APATH;/usr/local/bin/a+"}
```

or

```
$(unset ATREE APATH;/usr/local/bin/a+)
```

2. The Emacs Programming Environment

This chapter gives the standard A+ key settings and a brief summary of some Emacs functions. See an Emacs manual for more advanced Emacs functions. "The Window Manager", in "User Interactions with Displays" is also relevant to Emacs, and so is "Invoking A+".

Key Press Notation

C-x
: Hold down **Control**, then press **x**.

C-x C-b
: Hold down **Control**, then press and release **x**, then press **b**.

C-x k
: Hold down **Control**, then press **x**, then release both, then press **k**.

Emacs Key Functions

Key Sequence	Function
Backspace	Delete the previous character.
C-/ or **Undo**	Undo. Used immediately after entry of an A+ expression, removes its output.
C-a	Move the cursor to the beginning of the line.
C-d	Delete character. Also see **Delete**.
C-e	Move the cursor to the end of the line.
C-g	Cancel the partial entry in the minibuffer.
C-h	Help.
C-k	Kill line: delete line or part thereof and put it in the kill ring. See **C-y**.
C-r	Incremental search backward.

C-s	Incremental search forward.
C-x C-b	List information about all current buffers.
C-x C-f	Goto file: get the file (named in minibuffer) in a new buffer.
C-x C-s	Save the buffer in the file named on the status line (usually where it originally came from).
C-x C-w	Save the buffer in the file whose name is entered in the minibuffer (command line).
C-x k	Kill (delete) the buffer.
C-y	Yank: insert the last entry of the kill ring. See **C-k**.
Delete	The default is to delete the previous character (sigh), but it can be set to the command `delete-char` using `global-set-key`.
R1 or **F5**	Show one window in the screen.
R2 or **F6**	Show two windows in screen; if there are two, move the cursor to the other window.
R3	(No function set.)
R4 or **F7**	Next buffer (cycle through buffers).
R5 or **F8**	Previous buffer (toggle buffers).
R6	Go to (enter line number in minibuffer).
R7	Top: the first line of the buffer to the top of the window.
R8 or **F11**	Move the cursor up one line.
R9	Scroll up to earlier lines, with a two-line overlap.
R10	Move the cursor backward one character.
R11	Center the line with the cursor in the window.
R12	Move the cursor forward one character.
R13	End: move the cursor to the end of the buffer.
R14 or **F12**	Move the cursor down one line.
R15	Scroll down to later lines, with a two-line overlap.

Emacs A+ Mode Key Functions

Key Sequence	Function
C-c C-b	Load buffer. Like loading a script file, but without the need to save the buffer.
C-c C-c	Interrupt the A+ process. (From a shell, not Emacs, A+ is interrupted by a single **C-c**.)
C-c C-k	Kill (terminate) the A+ process. If this doesn't work, try `kill -9 pid` in an XTerm.
C-c C-m or **C-c Enter**	Get the previous input line.
C-c C-o	Kill (flush) the last output, and append "`rrr output flushed rrr`" to the input that elicited it.
C-c C-p	Go to the last input line.
C-c C-q	Quit the A+ process.
C-c C-s	Move the beginning of the last output to the top of the buffer and place the cursor there.
Enter	If the cursor is on the bottom line, execute that line; otherwise, send the current line to the bottom, for possible editing and execution. Ignore *leading* asterisks and any blanks following them, presuming that they indicate suspension or unbalanced punctuation, and the prompt; not exponentiation, or asterisks and blanks within a quoted character string.
F1	Help.
F2	Load and execute line (in the A+ log, like **Enter Enter**). In the log, *but not in a script*, ignore *leading* asterisks and any blanks following them, presuming that they indicate suspension or unbalanced punctuation, and the prompt; not exponentiation, or asterisks and blanks within a quoted character string.
F3	Load the program (function or operator) in which the cursor appears, in a script or the log. **Caution:** The program is read exactly as it appears, including any asterisks indicating continuation in an A+ session log, which will *not* be interpreted as such. For **F3**, the buffer must be in a-mode. To set it, enter **Esc-x a-major-mode**.

F4	Go to the buffer named "*a*" if it exists, or create one, and start up A+ if that buffer has no process. The release can be set in Emacs by **Control-c a**, as described at the beginning of "Invoking A+".
F5	Show one window in the screen, the one the cursor is currently in.
F6	Move to the other window in the screen, creating it if it doesn't already exist.
F7	Next buffer (cycle through buffers).
F8	Previous buffer (toggle buffers).
F9	Enlarge the window.
F10	Shrink the window.
F11	Scroll down to later lines, with a two-line overlap.
F12	Scroll up to earlier lines, with a two-line overlap.

The Meaning and Default Values of Emacs Lisp Variables

Variable	Meaning	Default Value
a-host	The name of the host machine on which to run A+.	nil
a-log	The name of the log file for A+ sessions.	`"~/.emacs_a"`
a-mbytes	The starting workspace size in megabytes.	4
a-mbytes-threshold	The largest a-mbytes value to be honored without a question.	16
a-plus-rest	Additional argv elements, to go after the `-w` parameter.	
a-prog	The name of the A+ program (release) to run; see "Invoking A+", regarding the form that A+ release names take.	`"/usr/local/bin/a+"`

To see or set the A+ mode (a-mode) Emacs variables, press **Esc x** when in Emacs, and enter **`a-options`** in the minibuffer following the `M-x` prompt. A new buffer will appear, containing a three-line paragraph for each variable and empty comment lines (two semicolons, followed by blanks) in between. "Key Definitions in a-Options Buffer" shows special key definitions in this buffer. When done, the buffer need not be saved; simply press **Control-x k**.

Key Definitions in a-Options Buffer

Key	Definition
n or **space**	Move to the next variable definition.
p	Move to the previous variable definition.
s	When the active line contains a variable value, move the active line to the command line (minibuffer) and prompt for a new value. Note that the quotes must be included in character values.
x	Toggle the value of the variable between the boolean values t and nil.
1	Set the variable to boolean value t.
0	Set the variable to nil.

3. Workspaces and Scripts

Parsing Rules for A+ Script Files

A+ script files are parsed from top to bottom when loaded, and all unresolved names are assumed to be variables. For example, suppose the following two lines appear in a script, in this order:

```
f{x}:g ⊤ x
g{y}:y,' '
```

When this script is loaded, the definition of the function *f* is parsed before that of *g*, and therefore *g* is assumed to be a (global) variable in the definition of *f*, and the symbol ⊤ is taken to be the dyadic primitive. Then the definition of *g* is parsed and it is determined that *g* is a function. When *f* is evaluated, a type error results because the dyadic primitive ⊤ is passed a function as a left argument.

```
     f`abc
 ⊤: type
*     ∨x          ⍝ Inquire about x
 `sym             ⍝ It's a symbol.
*     ∨{g}
 `func            ⍝ But g is a function.
*     →
```

This problem could have avoided by defining *g* before *f*. In general, however, and certainly in large applications, it is not reasonable (and sometimes not possible) to maintain scripts so that functions are always defined before they used. The problem could also be avoided by using the alternate function-call syntax; in the above example:

```
f{x}:g{⍉ x}
```
Now the A+ process can tell from the syntax that `g` is a monadic function, whether or not it has been defined. This syntax is unambiguous, whereas that of `f` in the original definition is ambiguous.

Workspaces

Expression Continuation in Entry and Function Definition

When an expression, expression group, or function definition is entered on several lines in an A+ session, A+ supplies stars at the left of each line to indicate the depth of punctuation. In general, these stars are used to indicate the total number of open quotation marks (at most one), braces, and parentheses, and an expression or function definition remains open until they are closed. Brackets are not treated in this way, however, and in the absence of an open quote, brace, or parenthesis a line or series of lines with an open bracket elicits an error message. Pressing the **Enter** key while inside a quotation mark inserts a newline character in the quoted string, but pressing **Linefeed** causes A+ to ignore all entries on the current line that follow the open quote (including the **Linefeed** itself).

If any brackets, braces, or parentheses are open when a function definition is closed, a token error is reported and the definition does not take effect. At the time when such punctuation becomes isolated (e.g., after the closing parenthesis for the bracket in `(3 4;b←d[1;3)`), a message "Mismatched parentheses" (no matter what the punctuation - brackets, braces, or parentheses!) appears at the bottom of the screen in Emacs.

The expressions `case(target)` and `if(condition)` and `while(condition)` are each followed by an expression or expression group in actual usage, but they are considered by the parser to be complete (with a Null as the following expression) when they are followed by the **Enter**, so don't break a line there unless you have something else holding the expression open, and don't immediately follow an `if(condition)` with an `else` with the intention of returning and filling in the `if` action unless you are very sure you won't forget.

Function definition has one continuation peculiar to itself: if the **Enter** key is pressed immediately after the colon following the function header, the function definition remains open, for just one more line (unless that line produces a continuation). If anything, however, even just a space, is entered after the colon, then there is no implied continuation. In particular, if only spaces are entered after the colon, a parse error is reported and the opening of the definition has no effect.

Expression Groups for Immediate Execution

In immediate execution, when there is no pending punctuation or function definition, several expressions can be entered on one line, separated by semicolons. They will be treated as an expression group. The next section discusses the implications of this treatment for display.

One useful aspect of this treatment is that one can bring into the workspace for immediate execution a script line which is only part of an expression group. Putting it another way, an expression group or part thereof can be brought in and executed immediately as long as its punctuation is matching - as long as it contains either both or neither of the group's enclosing braces.

An expression in braces, with no semicolon, also constitutes an expression group.

Immediate Execution Display

When an expression or expression group is entered for execution, by pressing the **Enter** key, an error message may be displayed. If it is not, then the result is displayed, except that it is not displayed for an expression whose last operation is Assign.

The display of the Null is indistinguishable from no display, so in effect the displaying of the result can be suppressed for an expression or expression group by appending a semicolon to it, making the expression a courtesy expression group. Likewise, display can be forced for an expression alone on a line, even when it ends with Assign, by preceding the expression with a semicolon. (Preceding such an assignment in immediate execution by a ⍎, thus applying Print to it, causes it to be displayed twice, once by Print and once as the default display of an expression whose last function is not Assign.)

For function and dependency definitions, there is no additional display: the entered definitions, together with the A+ prompts indicating depth of punctuation, are the entire displayed response.

Default Display of Arrays

Arrays are listed in ravel order. Simple character arrays start at the left margin. Simple numeric arrays are indented one space. Numbers that are in columns in the display are aligned. Simple vectors are displayed horizontally. Simple arrays of higher rank are displayed with vertical columns, horizontal rows, one skipped line between cells of rank 2, an extra skipped line between cells of rank 3, and so on. For example:

```
      ⍳2 2 2 6
 0  1  2  3  4  5
 6  7  8  9 10 11

12 13 14 15 16 17
18 19 20 21 22 23


24 25 26 27 28 29
30 31 32 33 34 35

36 37 38 39 40 41
42 43 44 45 46 47
```

A < is used for a function scalar, since it must always be shown as an enclosed function expression. Furthermore, vectors of function scalars are displayed vertically. For example:

```
      <{+},<{-},<{=}
<  +          ⍝ Vector displayed vertically, with < showing enclosure of the
<  -          ⍝ function expressions, to make simple, depth-0 function
scalars.
<  =
```

Each nested component of a nested array begins on a new line. A < character is used to indicate the beginning of an enclosed object. It is also used sometimes to indicate a symbol, as shown above, but it is not so used in this manual except for that example. For instance:

```
      (1 2 3; 2 2⍴25; (3 4;⍳2 3;<<100))
<  1 2 3
<  25 25
   25 25
< <  3 4
  <  0 1 2
     3 4 5
  < < <  100
```

Resuming Execution

When execution of a expression or function has been suspended because of an error or an interruption, expressions can be entered for immediate execution. (The expressions may include calls of other functions and may lead to suspensions of their own.) Values, shapes, types, etc. of both global variables and ones local to the suspended function may be examined and changed. The arguments on the stack (`&0 &1 ...`) can also be examined and changed. (Changing variables which were arguments to the suspended function has no effect on the stacked arguments.)

To resume execution of the suspended function, with the workspace size increased if necessary, enter

```
←
```

alone on a line.

To abandon execution, and execution of any pendent function that is waiting on this suspended function, enter either

```
→
```

or

```
$
```

alone on a line.

To abandon execution of the last *n* suspended functions, enter

```
$reset n
```

and to abandon execution of all suspended functions, enter just

```
$reset
```

See the Reset command.

The Form of Error and Other Messages

Error reports are prefixed by a comment symbol and the type of report in brackets, to make it more explicit and also to prevent it from spoiling a part of a log that would otherwise be usable as a script:

```
     3+0÷5-5
⍝[error]  ÷: domain
⍝[error]  segv
```

and

```
⍝ PKG: 1: Storing .two      ⍝ See the package example.
```

There are, however, some error messages that are not in this form, none of them strictly A+ error messages. Among them are

```
filename: No such file or directory
not an `a object
```

and a great many messages from adap.

Contexts

To help in avoiding name conflicts, A+ provides contexts. A context provides a level of qualification for global names. The fully qualified form of any global name is *c*.*x*, where *x* is the unqualified name of the object and *c* is the context for that name. Every name is accessible from every context by the use of its fully qualified form.

The workspace always has one current context. Initially, at the beginning of an A+ session, it is the *root* context, whose name is the empty string. So `.y` is the form that a fully qualified name

takes in this context. Such a name always refers to the root context. The current context can be changed by `$cx cxt` as discussed for the Context command; by convention, `$cx .` is used to change the current context to the root context - i.e., the context name is given as a period. In an expression entered for immediate execution, any unqualified name, no matter what its use - reference, specification, call, definition -, is understood to be implicitly qualified by the current context. Thus unqualified names can be used for objects whose full names are qualified by the current context.

A function also has a context. If a fully qualified name was used for it in its definition, then the context is the one referred to in that name; otherwise, the function's context is the one that was current when it was defined (and indeed, its fully qualified name includes the name of that current context). All unqualified global names within a function are implicitly qualified by the function's context; the context in which the function is executed is irrelevant to their interpretation. When a function is suspended, the context for immediate execution is the function's context. If the context is changed while a function is suspended, resumption of execution will automatically change the context back. Completion of execution returns the context to what it was when the function was called.

Local and Global Names

To avoid clutter in the workspace and to reduce the possibility of name conflict, a name can be local - i.e., have meaning only in the function in which it occurs and disappear when execution of that function is complete. Such a name must be unqualified, and it is made local by an Assignment within the function in which only the name appears to the left of the arrow. All qualified names are global. See "Scope of Names".

A function can see only global names and its own local names. A user can see only global names and the local names of the latest suspended function (and not pendent ones).

Visible Use of a Name

A visible use of a name occurs when the name appears in A+ code directly, and not in a character string or a symbol. A use that is not visible is an implicit reference, which occurs through the employment of Execute (`⍎`) or Value (`⍕`). This distinction is important for dependencies, which are invalidated only by visible uses of names (cf. "Dependencies", especially "Evaluation" and "Dependencies Defined").

Listing Names

System commands and system functions are provided for listing names of various kinds in the workspace:

- Commands described in "System Commands" include:
  - Commands
  - Context (gives context name)
  - Contexts (all contexts in the workspace)
  - Dependent Object Names
  - External Functions
  - (defined) Functions
  - Global Objects
  - (defined) Operators
  - System Functions, and
  - (user) Variables.
- Functions described in "System Functions", include:

- All Dependent Object Names
- Dependent Object Names, and
- Name List (user variables, functions and operators, dependencies, external functions, system variables, functions and commands, and contexts, as well as keywords and symbols).

Reporting the Environment in the Active Workspace

A number of system functions and commands are available for inquiring about the active workspace:

- "System Commands" describes:
  - Callback Flag
  - Current Context
  - All available Contexts
  - Debugging State
  - Execution Suspension Flag
  - Input Mode
  - Printing Precision
  - Protected Execute Flag,
  - State Indicator,
  - Stop
  - Terminal Flag
  - Version
  - Workspace Available, and
  - X Events Flag.
- "System Functions" describes:
  - Get System Variable
  - Hash Table Statistics, and
  - Work Area.

See also "Listing Names", above.

4. Files in A+

Script Files

As described in earlier chapters, the Load and Load and Remove system functions and the Load and Load and Remove commands deal with script files, loading such a file into the active workspace by interpreting every one of its lines, starting at the top, essentially as if the lines had been entered directly in the active workspace. The difference is that after a file is loaded, the current context and the current directory are the same as they were when the Load command or function was initiated: each is automatically restored if it was changed during execution of the lines of the file. See "Workspaces and Scripts" for more details concerning this kind of file.

Mapped Files

Data files in A+ are called *mapped files*.

From the viewpoint of the A+ primitive functions, a mapped file is (for the most part) an ordinary array when accessed through an associated variable. A mapped file can always be referenced and (if opened for writing) selectively assigned (see "Selective Assignment") as if it were an ordinary array.

Ordinary (as opposed to selective) Assignment conveys not only value (including shape) but also mapping status to its target. In particular, a target that is mapped will remain so if and only if the righthand side of the Assignment is mapped (see bullet items below); otherwise, it will become an ordinary array. Also, since passing arguments and results acts like ordinary specification, if a function is called with a mapped file as an argument, that (local variable) argument in the function will be mapped, and if the result of the last expression executed in a function is mapped, so is its result.

- Primitive functions whose results for mapped arguments are always identical in value and *mapped*:
  Left (⊣), Print (↓), Result (← with no left argument), Stop (∧).
- Primitive functions whose results for mapped arguments are perhaps not identical but always *mapped*:
  Disclose (⊃), Enclose (⊂), Raze (⊃).
- Primitive functions whose results for mapped arguments are always identical in value but *unmapped*:
  Right (⊢), Identity (+).
- Primitive functions plus arguments whose results for mapped arguments are identical and *mapped*:
  `m[]` `()#m` `()⊃m` `(⍳0)⊃m`
- Primitive functions plus arguments whose results for mapped arguments are identical and *unmapped*:
  `m[;]` `(;)#m` `1/m` `(∨m)∨m` `(∨m)∨⍨m`

A+ provides a special syntax that is particularly useful for updating all elements of a mapped file:

```
a[]←b
```

and a special syntax that is particularly useful for appending new items to a mapped file:

```
a[,]←b
```

Again, see Selective Assignment for definitions of these expressions. Appending items to a mapped file with the latter special sequence is permitted only when enough space already has been allocated for the new items. If not enough space has been allocated, a maxitems error is reported. Such allocation is accomplished with `_items` ("Items of a Mapped File"). After allocation, the file must be remapped.

On AIX, the workspace size can't be enlarged when there is a file mapped (`⍴1 0⊃_dbg{`display;`beam}` is nonzero), because A+ requires that the entire workspace be contiguous and AIX requires that all space be taken from one end of the address space; thus, the mapped file blocks workspace enlargement.

When one mapped file is created from another, by ordinary Assignment, the new mapped file has only enough space to contain the data; the value of `_items{¯1;}` for the old file is not carried over to the new file.[1]

Mapped files can be shared by different A+ processes, and all A+ processes on the same machine will immediately see any updates by other A+ processes on that same machine. A+ processes on a different machine, however, may not see the updates immediately, or ever. Whether or not the updates are seen depends on when and if the underlying operating system refreshes certain virtual memory pages. They may be only partly seen when items do not correspond to pages.

When a mapped file is an argument to a defined function, the corresponding local variable is, of course, also a mapped file. If that argument is a global variable, then both variables, global and local, refer to the same file, and therefore can affect each other's values if they have write access to the file. If they have read access only and one variable is used for writing, then it is no longer a mapped file, but this change in its status naturally does not affect the other variable. In short, at the point at which the function is called they are two independent variables referring to the same file.

To obtain just the value of a mapped file, i.e., a copy of the file, the Right function can be used. See the examples below.

The *I* primitive function (Map or Map In, or Beam) is used to create, reference and update mapped files.

Error Messages

If a Map operation fails, you should get a domain error. A more specific error message appears in the session log if one is generated by Unix itself. (The message is not guaranteed to appear, but Unix so reports most problems.) If the Map is within a protected do, the do result shows the domain error, not the specific error. To help determine the cause of the failure, you can use `sys.errno{}` to retrieve the Unix error number, although exactly which system call failed will sometimes be ambiguous.

If an "Operation would block" (or perhaps "interrupted system call") error message is received, it usually means that there has been a temporary network outage or extreme slowdown, so the map operation timed out. Before reporting this problem to the user, A+ makes repeated attempts to carry out the operation, over a period of about a minute. Be aware that such a delay can occur.

If a "not an `a object" error message is received, it may mean that the address space for all mapped files (including atmp) has been exhausted, and that therefore the file header does not agree with the file body. Mortgage files can be especially large, and may contribute to the exhaustion of this 1Gb to 4Gb space.

If a file opened for reading only is "wrong-endian", it is copied into a variable in atmp and a warning message is issued. For writing, including local writing, Beam rejects a "wrong-endian" file.

Concurrent Reading and Writing of a Mapped File

When two users, A and B, say, map the same NFS file (as A+ variables), A for writing and B only for reading, what B sees when A makes a change depends upon the location of their machines and the file in the network:

1. A and B are on the same machine and A updates the file in place. B will see the change immediately, regardless of whether B mapped the file before or after A made the change.

2. A and B are on the same machine and A uses `_items` and appends to the file before B maps (or remaps) the file. B will see the change immediately.
3. Like 2, but A uses `_items` after B maps the file. There is no problem unless A has created a new page and B attempts to reference the new page, in which case B will get a segv error.
4. Like 1, 2, and 3 except that A and B are on different machines. The results are more or less the same as in those three cases but may be delayed. Unix looks for a page in its cache first, and uses it if it is there. Depending on system activity, a page may remain in the cache even after it has been changed on another machine, as NFS has no way of knowing that the page has been changed. Hence B will see A's change only after the affected old pages are removed from the cache by the system to make way for more recently referenced pages. B can get the new pages by remapping the file before referencing it (and after A's changes), unless A has engaged in the bad practice of changing the file on a machine other than the one on which it resides - in which case any changed pages will have to make their way via NFS to the host machine for the file before users on any other machines see them.
5. A and B and the file are all on the same machine and A rewrites (remaps) the file. The system creates a new file (a new "inode") and gives it the old name. If anyone has the old copy of the file opened or mapped (the reference count is greater than zero), the system keeps the old file around with the old inode. When B references the mapped variable (without remapping), the reference is by inode and therefore to B's private copy of the old file; A's newly written file is not seen. These old copies take up space on disk and remain until no user has them opened or mapped.
6. Like 5 except that but B is on a different machine from A and the file, and suppose enough activity has occurred after A rewrote the file to flush the cache in B's machine. A reference to the mapped file by B will produce a "Stale NFS file handle" error report if no one on A's machine has happened to keep the file's reference count above zero and thus preserved the old inode. The system in B's machine has gone looking for an inode that no longer exists (inodes are unique with a domain).
7. Like 6 except that A renames the file before rewriting (remapping) it (under its original name). Then B sees the old file, as it was before A's actions, even if enough activity has occurred to flush the cache in B's machine, because the old copy of the file, under the old inode, is still around. (For example, files in the mas database that get rewritten are first renamed.)

Warning! An application can crash, with a bus error, if two or more users are writing (not necessarily at the same instant) in the same mapped file. The mapped file mechanism does not mediate independent updates.

Mapped Files On Remote Machines

Using mapped files across NFS or AFS is problematic for anything other than a simple read, and sometimes even for that. A job which will run on one machine and access data in mapped files on another machine should probably have a server process on the machine where the files are. The application should then submit queries and updates and receive data through an adap connection to that server.

Map *y*⍎*x*

Arguments

y is an integer and *x* is a symbol or character string, or *y* is a symbol or a character string and *x* is a simple character or numeric array.

Definition

There are two cases.

- If *y* is an integer, then *x* names a mapped file. The argument *x* can give a path to the file as well as the file name. If no path is given, the environment variable APATH is referenced for the search path. If the file name ends with the suffix "*.m*" the search is simply for a file with that name. Otherwise, the suffix "*.m*" is appended to the name and a file with that augmented name is sought; if the search fails, then a file with the original name (without the "*.m*") is sought. (Notice that this search order is opposite from that for the Load command, which applies suffixes only if the search using the given name fails.) In this case (*y* an integer), *y* can have one of three values:

  0⍎*x* Open the file for reading only. Writing in a file opened for reading only (!) is treated like ordinary assignment: the data to be written is inserted in a *copy* of the mapped file, which is thereafter treated like an ordinary array.

  1⍎*x* Open the file for reading and writing. The system function *_items* is used to allocate space for the file; see the examples below. In AIX, this function causes the timestamp to change even if you make no changes to the file, because A+ overwrites the first four bytes as part of the beam-in procedure.

  2⍎*x* Open the file for reading and local writing. Writing in a file opened for only local writing causes the overwritten pages and the first page, which contains the header, to be written as if part of an ordinary array, while pages not overwritten through this variable are read from the file. In other words, when a process uses 2⍎... access, the resulting mapped-file variable cannot change the file that others see, but there is the efficiency of sharing pages that the process has not changed and the advantage of seeing any changes that were made via 1⍎... access to these pages by other processes. *Warning:* Items may not coincide with pages, so with 2⍎... access you may see only part of someone else's change, even on the same machine.

- In the second case for this primitive, *y* is a symbol or character string naming a file and *x* is a simple character or numeric array; the effect is to create the mapped file named in *y* with the value *x*. If the file name in *y* ends with "." followed by anything or nothing (unlike the first case, which treats only "*.m*" specially), then it is used as the name of the created file; otherwise, the name of the file is the value of *y* with "*.m*" appended.

When a file is mapped, the Unix command sequence is `open(); mmap(); close()`. `close()` does not unmap the file but does free the file descriptor. When the file is unmapped - by assigning a new value to the variable, expunging the variable, ending the A+ session, or whatever - `munmap()` is called.

Map In ⌶x

Argument

x is either a symbol or a character vector.

Definition

⌶x is equivalent to 0⌶x.

Examples

```
     fro←0⌶`file    ⍝ Create a mapping of the file file.m for reading only.
                    ⍝ If fro is changed, it becomes an unmapped array.
     frw←1⌶`file    ⍝ Create a mapping of file.m for reading and writing.
     frlw←2⌶`file   ⍝ Read and local write: see others' changes; don't show
own.
     frw[i]←new     ⍝ Changes in place to frw also modify the file file.m
     frlw[i]≡new
 1                  ⍝ Change seen.
     frlw[i]←newer  ⍝ This change will not modify file.m.
     frw[i]≡newer   ⍝ Only frlw was modified:
 0                  ⍝ new and newer are unequal.
     `file⌶array    ⍝ Write a simple array as mapped file file.m

     'test.m'⌶0 4⍴0   ⍝ Create a file for a matrix with 0 rows.
     ⍝ Make the allocation 10 rows. Left argument is the total number of
items in the file.
     _items{10;'test.m'}
 0                  ⍝ Successful allocation: result is the former number of
items.
     t←1⌶'test.m'   ⍝ Map the file test.m in read/write mode.
     ⍴t
 0 4
     t[;]←10 20 30 40     ⍝ Append a new row.
     ⍴t
 1 4
     t[;]←1 2 3 4         ⍝ Append another new row.
     ⍴t
 2 4
     t
 10 20 30 40
  1  2  3  4
     _items{¯1;'test.m'} ⍝ Determine number of allocated items.
 10                      ⍝ Still 10 allocated.
     ⍝ Make the allocation 20 rows. Left argument is the total number of
items in the file.
     _items{20;'test.m'}
 10                   ⍝ Successful: result is former number of items.
                      ⍝ Since _items was executed, remap the file test.m
     t←1⌶'test.m'
     a←1⌶`file        ⍝ a is a mapped file.
     b←a              ⍝ b is a mapped file.
     c←⊢a             ⍝ c is not a mapped file, but has same current value as
a
     f{x;y}:{...}
     f{a;⊢a}          ⍝ In this invocation of f, arg x is a mapped file and y
is not.
```

Unix Text Files

A simple way to read a Unix text file f is to enter

```
m←sys.readmat{f}
```

where *f* is a character vector giving a path name. The file is read into *m* as a matrix, in which all rows have been made equal in length to the longest row in the file, by appended blanks.

A file can be read as a vector, with newline characters embedded, by a Pipe In command like

```
$<FileVarName cat FileName
```

or by a function such as:

```
read{file}:{
     if (0>s←sys.filesize{file}) ⍎⊥'read failed: ',⍕s;
     a←s⍴' ';
     fd←sys.open{file;`O_RDONLY;0};
     sys.read{fd;a;s};
     sys.close{fd};
     a
     }
```

Partition Count and Partition (⊂) can make the result of such a vector into a nested array of lines.

In this function the local variable *s* tells *sys.read* how many characters to read. Clearly, with minor alterations to the code shown, the file can also be read a portion at a time. Moreover, the function *sys.lseek* can be used to choose a point from which to start reading a portion of the file.

Data can be written to a Unix text file by Pipe Out and Pipe Out Append commands like

```
$>FileVarName FileName
$>>FileVarName FileName
```

when the data is a character vector with embedded newlines, or by functions like the following, which also convert any character matrix arguments to vectors, deleting trailing blanks and appending newline characters as required:

```
write{file;data}:{
     if ((`char=⍬⍴data)∧2=⍴⍴data) data←clean{data};
     if (0<fd←sys.open{file;
                       `O_CREAT`O_TRUNC`O_WRONLY;
                       8⊥6 4 4})
        {
        sys.write{fd;data;#data};
        sys.close{fd};
        };
     }

clean{n}:⊃(cleanline¨<@1 n),¨<"\n"

cleanline{x}:(-+/∧\' '=⌽x)↓x
```

You can easily devise variations for yourself, for both reading and writing. For more details, such as the meanings and permissible values of the arguments to the sys functions, see "The sys Context".

5. Timing A+ Functions

Time `time s`, or `time()`, or time expression, or `time{}`

Argument and Result

In the four expressions above, the argument is, respectively, *explicitly* one or more scalar symbols (not an expression whose *value* is a scalar symbol or vector of scalar symbols), the Null expressed as a pair of parentheses, some other expression, or absent. The result is a vector that is numeric, symbol, nested, or Null.

Definition

The effect of `time` is to set or reset a timer, or to return the results of a timer. All timing information is expressed in milliseconds. In the descriptions that follow, *system time* refers to the cpu time spent in system calls made by the A+ processor, *user time* refers to the cpu time spent by the A+ processor directly, and *elapsed time* is clock time (you will notice it increasing even when you are doing nothing).

When you are timing any code whose execution time is even moderately short, you should use a dyadic do statement to execute the code many times, and take an average. Doing so mitigates any imprecision in the timing and allows for circumstances, like timeslicing, that can vary for the same code.

Timings will generally be different for the same code with different data, being affected by type, size, nesting, and so on. One method of accomplishing something may have a substantial fixed cost but be relatively insensitive to the size of the data, whereas another may have little fixed cost but a strong dependency on size. You should time the various cases that can be expected to arise, so that you can determine which method is best for your expected distribution of cases. It may turn out that you will want to include several methods and choose among them during execution.

`time` is not a true function, although it has functionlike attributes. Specifically, if `time` were a true function then the two expressions `time{}` and `time()` could not both be syntactically correct, because the former indicates that `time` is niladic and the latter monadic. However, both are valid A+ expressions. See the table "Timer Expressions and Their Meanings" for a description of all valid timer expressions. Although its name does not start with an underbar, `time` is always available, like a system function.

Timer Expressions and Their Meanings

Expression	Effect	Result
`time` *expression*	Time the indicated A+ *expression*.	3-element numeric vector of user, system, elapsed time used to evaluate *expression*.
`time` *s*	Set the timer for the functions named by the symbols in *s*. Any external and system functions named in *s* are ignored. If a function is redefined, the timer switches to the new definition.	*s*, a vector of symbols. Including external or system functions in *s* may cause a bus error in some releases.
`time{}` following `time` *s*	Return timer results.	2-element nested vector `(`*s*`;`*matrix*`)`, explained below.
`time{}` otherwise	Return timer results.	3-element numeric vector of user, system, elapsed time since the current session began.
`time ()`	Reset the timer, i.e., undo the effect of `time` *s*.	Null.

Result (s; matrix) of time s Explained

Expression	Meaning
matrix`[`*i*`+1;0]`	The number of times the function named in `i#s` was called.
matrix`[`*i*`+1;1]`	Total user time spent in the function named in `i#s` exclusive of others being timed.
matrix`[`*i*`+1;2]`	Total system time spent in the function named in `i#s` exclusive of others being timed.
matrix`[`*i*`+1;3]`	Total elapsed time spent in the function named in `i#s` exclusive of others being timed.
matrix`[0;0]`	0 (meaningless - just filler).
matrix`[0;1]`	User time spent outside all functions named in *s*.
matrix`[0;2]`	System time spent outside all functions named in *s*.
matrix`[0;3]`	Elapsed time spent outside all functions named in *s*.
`+/`*matrix*`[;1]`	Total user time since `time` *s* was executed.

`+/matrix[;2]`	Total system time since `time s` was executed.
`+/matrix[;3]`	Total elapsed time since `time s` was executed.

Examples

```
    f n:{n do 3;}                    ⍝ Don't need to see explicit results.
    m g n:{m do f n;}
    time `f `g                       ⍝ Set timers for f and g
    100 g 10                         ⍝ Call g and, from g, call f
    1000 g 5                         ⍝ Call g again.
    10000 do 2+3                     ⍝ A call outside g
    time{}
<   `f `g
<      0  160     0 7080     ⍝ 160 ms user, 0 system, 7080 elapsed not in f or g
   1100  100   110  230     ⍝ f: 1100 calls; 100 ms user, 110 system, 230
elapsed.
      2   90   130  250     ⍝ g: 2 calls; 90 ms user, 130 system, 250 elapsed.

    time ()                          ⍝ Clear timer.
    time{}
 500 580 20110       ⍝ 500 ms user, 580 system, 20,110 elapsed from the
beginning.

    sf←(`a`b`c`d`e`f`g`h`i `j `k `l `m `n;
        (1;2;3;4;5;6;7;8;9;10;11;12;13;14))
    s←1000⍴(0⊃sf),`o    ⍝ Includes an invalid index for the slotfiller just
defined.

    time (j←999) do r← 0⍟'s[j]⊃sf'
 810 100 920      ⍝ Times for avoiding index errors by an ⍟ Protected Execute.

    time (j←999) do r← do s[j]⊃sf
 170 70 240       ⍝ Times for avoiding index errors by a do Protected Execute.

    time (j←999) do r← {i←(0⊃sf)⍳s[j];if (i<#0⊃sf) i⊃1⊃sf}
 300 0 310        ⍝ Times for avoiding index errors by mimicking symbolic
indexing.
```

6. Interprocess Communication: adap

The adap context is native to A+; there is no need to load adap. "idap.+" is loaded automatically when the interpreter is started. idap.+ installs backward-compatible behavior on top of the automatically loaded i context, which uses the MSIPC class of MStk.

DAP (Distributed Analytics Platform), a set of tools supporting interprocess communication, has been used extensively in non-A+ applications. The functions in the adap context are a layer on top of DAP, providing application writers with the means to:

- establish and maintain communication between pairs of processes, such as a client-server pair;
- specify the format of the data to be interchanged by a communicating pair of processes; and independently, to
- establish and maintain timer events.

Both asynchronous and synchronous interactions are supported. In asynchronous interactions using the A protocol (cf. "Protocols for Sending and Receiving Data", and especially "The A Protocol"), both single and burst modes are supported. These tools are in the adap context, which is always available in an A+ session.

Interprocess Communication

Communication based on adap is carried out between two processes - A+ processes or not - , but not directly among more than two. The mode of communication can be either synchronous or asynchronous. In synchronous communication, one partner sends a block of information to the other and is blocked from further action until it receives a response. In asynchronous communication, one partner sends information to the other and then simply goes about other business; if there is a response to what was sent, it is processed whenever it arrives. Thus one partner might send several independent requests, each requiring a response, before the response to any one of them is received. When one process is communicating asynchronously with several others and receives messages from more than one of the them, there is no guarantee that they are received in the same order they were sent. Only this is certain regarding the order of events in asynchronous communication: if a process sends more than one message to one particular partner, the messages will be received in the order in which they are sent.

At first thought, synchronous communication may appear to be somewhat easier to manage than asynchronous, but in general it is not, and it is of less general use. Synchronous communication can be used most effectively when one partner has nothing else to do but wait for the response from the other. Likewise, asynchronous communication may appear to be more complicated because messages are processed whenever they arrive, and there is no way to know when the arrivals will occur, or - when there is more than one source of messages - in what order. However, adap's way of managing asynchronous communication, namely events and callback functions, is quite straightforward.

The rules for interprocess communication are strictly up to the partners. It can be agreed that one partner always processes requests from the other; that either partner can initiate a request of the other; that either partner can send a message at any time; that not all requests require a response; and so on.

Establishing a communication channel is not a symmetric procedure; one partner is designated to listen for the other, while the other attempts to connect to the listener. The listener initiates its end of the communication channel by executing `adap.Listen`. The other partner executes `adap.Connect`. For convenience they will be called the listening partner and connecting partner, respectively. In principle, it doesn't matter which partner happens to go first[1]. If the listening partner goes first, the effect of executing `adap.Listen` is to continually listen for connecting partners; if the connecting partner goes first, the effect of executing `adap.Connect` is to continually attempt a connection until the listening partner starts listening. Once both partners have attempted to establish communication, the communication channel is formed.

Even though the listening partner continues to listen and the connecting partner continually attempts to connect to its partner until successful, both `adap.Listen` and `adap.Connect` return immediately after being executed, for otherwise the partners would not be free to do other work. The status of the underlying listen and connection attempt are monitored with callback functions on adap events.

The asymmetry in establishing communication is also reflected in the fact that the connecting process can link up with only one partner for each execution of `adap.Connect`, while the listening process can link up with many partners with just one execution of `adap.Listen`. Executing `adap.Listen` causes a listening port to be established, which remains open to receive connecting partners until the listening partner shuts it down. Executing `adap.Connect` causes a connecting port to be established, over which the connecting partner continually tries to connect to one and only partner, until either it is successful or the connecting partner shuts it down.

Client-Server Communication

Client-server communication is an important, commonly used mode of communication. It is the mode by which one partner, the client, typically takes the lead in communication with a server, by specifying the services to be performed; the server is the passive partner, waiting for requests from clients before taking action.

The server may provide a common service for many clients, such as real time data access, or a dedicated service, such as data base access and analytic computations. In the former case the server is the listening partner because of the one to many nature of its service; it is probably started automatically by some means of the operating system, and is always available. In the latter case, the client-server model may be used simply for load balancing; the server may actually be started by the client, and may be the connecting partner.

Callback Functions and Events

An event is an asynchronous change in the state of an application which can trigger an action by the application. There are two classes of events that are of concern in adap applications, interprocess communication events and timer events. Interprocess communication events mark significant changes in the communication state between processes, while timer events mark time interval expirations. The means by which an application program takes action is a callback function, which is automatically called when the event takes place.

For example, when a client requests a connection to a server, a callback function in the server application is automatically invoked so that the server can register the client. When a message from the client to the server is ready to be received, a callback function is automatically called so that the request can be processed. Similarly, when the message sent back to the client is ready to be received, a callback function in the client application is automatically called. In this manner a server can process random, or asynchronous, requests from clients, and a client can continue with other tasks, but still process responses from servers as they arrive.

Interprocess communication events fall into several categories, called *event types*. For example, the receipt of a message from a partner is an event of type `` `read ``. The callback function for the receiving partner's service handle (which identifies the connection) is called with arguments identifying the event type and the partner that the message is from. There is a small, fixed set of event types in adap (see the table "adap Asynchronous Interprocess Communication Event Types"); typically a callback function consists of a single case statement with a case for each type.

Callback functions are established for the partners when they call `adap.Connect` and `adap.Listen`. All adap callback functions have three arguments, and their meanings are always the same:

```
CallBackFunction{service_handle;event_type;call_data}
```

In interprocess communication, *service_handle*, which is a scalar integer, is the means by which communicating partners identify one another (see "Service Handles"); *event_type* indicates the general class of the action that caused the callback to occur (see table below); and *call_data* holds data associated with the action (see table below). In the case of event type *`read*, the associated message is *call_data*.

adap Asynchronous Interprocess Communication Event Types (Callback Function *CBF{service_handle;event_type;call_data}*)

Event Type	Description And Action To Be Taken
`choose	A choose event occurs when only *`name* is specified in a service descriptor argument to adap, and more than one service has a matching service name. In this case *call_data* is a vector containing a complete service descriptor for each matching service. The result of the callback function should be one of these service descriptors. Note: it is not always useful for a callback function to produce a result, but in this case it is.
`connected	A connected event indicates to both the listening and connecting partners that a connection has been made. For example, a server would typically add the client identified by *service_handle* to a list of clients and perhaps perform some initialization, while a client would start off the client-server communication by sending a message to the server.
`error	An error event indicates that something has gone wrong from which adap cannot recover. Unlike the reset event (see below), it is left to the application to recover from an error event: to call adap.Close on the handle and then either to restart if possible or to terminate gracefully. No attempt is made to qualify the type of error that occurred.
`read	A read event indicates that a complete message has arrived from the partner identified by *service_handle*, i.e., the entire message sent by the partner with one call to *adap.Send* has arrived. The data, i.e., the contents of the message, are in *call_data*. In burst mode (in the A protocol), all pending complete messages from the same service handle are included in *call_data*, each as an enclosed element; see rEventMode.
`reset	A reset event indicates to either partner that the connection between the pair has been broken, either by the partner closing his end of the connection, or by a recoverable system error. For example, a server would typically remove the client indicated by *service_handle* from the list of clients it is processing and close that service handle. A client would either explicitly close its service handle or, by doing nothing, allow the system to try to reconnect by automatically recalling *adap.Connect*. (The listening partner cannot maintain the old connection handle and wait for the connecting partner to reconnect, because the new service handle of the connecting partner will be different.)
`sent	A sent event indicates that one or more pending messages have been sent to the partner identified by *service_handle*. Messages are not necessarily sent

	immediately by *adap.Send*; some may remain pending and will be sent by adap at a later time. Messages that are sent immediately do not cause sent events. Rather, sent events occur when pending messages are actually sent. *call_data* is the number of pending messages sent. The total of the *call_data* values for all sent events plus the total of all results of *adap.Send* equals the number of messages sent to the partner.

Timers work a bit differently. Expiration is really the only event type for a timer. Consequently, if callback arguments for timers were the same as callback arguments for interprocess communication, the second argument for calls to timer callback functions would always have the same value, and so be useless. Therefore, timer event types have been defined to be the names of timers; a name is given when a timer is established (see *adap.SetTimer*). This means that timer event types are not a fixed set as in interprocess communication, but are determined by the application writers. In practice, of course, they are a fixed set within each application.

That the only true timer event type is expiration means that the callback function of a timer is called only when the timer expires.

The *service_handle* argument of a timer callback function is of use when the timer is to be closed or modified before it expires. The *call_data* argument is the time duration with which the timer was established (see *adap.SetTimer*), thus permitting the callback function to reset the timer with a related duration.

Each timer is established by calling *adap.SetTimer*, and its callback function is identified in that call. Consequently, each timer could have a separate callback function, but it is common practice to use one callback function consisting of a single case statement, where the cases are the names of the timers (see *adap.SetTimer*).

The adap Data Structures

Service Descriptors for Interprocess Connections

A service descriptor is a nested vector describing the connection a connecting partner is attempting to establish with the listening partner, and the connections the listener is willing to accept. The partners agree beforehand on the contents of this vector. The elements of the vector are arranged in attribute-value pairs; i.e., the vector is an association list. The first element of an attribute-value pair is the service attribute, and is one of *`host*, *`port*, *`protocol*, or *`name*. The second element of a pair is an appropriate value for the service attribute. These attributes can be set on a listener, and their values will then be given to any connection it establishes. (The *`retry* attribute is explicitly turned off and *`listener* is reference only.) See the "Service Descriptor Attributes" table.

For example, the following service descriptor uses the sample values in that table:

```
(`host;`s5; `port;9004; `protocol;`A; `name;`ThisService)
```

The order in which the attribute-value pairs appear is immaterial, although each attribute symbol must be followed immediately by its value. For example, the above service description is equivalent to:

```
(`name;`ThisService; `host;`bilbo; `protocol;`A; `port;9004)
```

If the name of the listening partner is registered in the operating system, it is only necessary to specify the name in the service descriptor; e.g. `(`name;`idn_apc)`.

Service Descriptor Attributes

Service Attribute	Description	Sample Value, Default
`` `host ``	The name, as a symbol scalar, of the host machine on which the listening partner runs. If both partners run on the same machine, then `` `localhost `` can be used. For `adap.Listen`, `` `host `` must be the local host; it can be given as its name, `` `localhost ``, or null.	`` `bilbo `` (not necessarily a real host name). *Default when anything other than* `` `name `` *is specified:* `` `localhost ``.
`` `listener ``	Reference only. This attribute is valid for connections. For those created by listeners, the value is the handle of the listener which created the connection. For "client" connections created through direct calls to `i.connect*` functions, the value of the `` `listener `` attribute is 0.	
`` `name ``	The name, as a symbol scalar, of the service. Some services are listed in the system yellow pages, and for them only their names need be given in service descriptors (see the `` `choose `` event type in the table "adap Asynchronous Interprocess Communication Event Types"). For others a complete service descriptor must be given, and the name serves only as an abstract identifier of the connection.	`` `ThisService ``
`` `port ``	The port on which the communication takes place. Port numbers are controlled by the system administrators. 0 means that adap should pick any available port; the number of the port chosen by adap will be the first element of `adap.getPort{servicehandle}`, and it is the application's responsibility to notify potential partners of it.	9004. *Default when anything other than* `` `name `` *is specified:* 0.
`` `protocol ``	The name of the format, as a symbol scalar, in which data is sent and received by the partners. See "Protocols for Sending and Receiving Data".	`` `A ``, for sending and receiving A+ arrays. *Default when anything other than* `` `name `` *is specified:* `` `A ``.
`` `retry ``	If 1, adap attempts to re-establish the Listen connection if the first try fails. If 0, it does not, perhaps the more	0

	reasonable course.	

Protocols for Sending and Receiving Data

A+ applications can use adap to communicate with other A+ applications and with non-A+ processes, in particular real-time data sources. There are different formats of the A+ data, or protocols, for the different types of partners: the A and simple protocols are used when both partners are A+ applications. In addition, there are the raw and string (as well as simple) protocols for sending or receiving unformatted character strings, and the ipc protocol for communicating with certain non-A+ processes that expect that protocol. The ipc protocol is not documented here.

The first contact with arriving real-time data is made by the line-reader processes, which format the input streams into discrete messages.

The raw protocol can be used to communicate with the lowest level of these processes, the line readers. Most A+ applications deal with the higher level protocols, but, as you can see, it is possible to establish communication at any level of the real-time data management scheme.

The A Protocol

The A protocol is for sending and receiving A+ arrays between A+ applications; it is specified in the protocol portion of the service descriptor as `` `A ``. These arrays cannot contain function expressions. The array is actually transmitted in CDR format (see Import an A+ Array, `sys.import`).

The simple Protocol

simple is designed as a fast, simple A+ protocol; it is specified in the protocol portion of the service descriptor as `` `simple ``. It is similar to the A protocol but does not accept nested, symbolic, or functional data. That is, it takes only simple integer, float, or character arrays, ones that can be mapped. The simple protocol does not use import or export (no CDR format), so an A+ object sent can be easily reconstituted in a C process. It does include a four-byte header containing the length of the A object in bytes.

The raw Protocol

The raw protocol is to communicate with non-A+ processes that send and receive character strings; it is specified in the protocol portion of the service descriptor as `` `raw `` or `` `RAW ``.

The string Protocol

string has more general appeal for communication between A+ and C processes than simple does. It is similar to raw, except that it includes a four-byte header with the length of the message, and ensures that only complete messages are delivered. (This differs from raw, in which a single send may come out as several reads, or vice versa.) It accepts (and delivers) only character vectors. It is specified in the protocol portion of the service descriptor as `` `string ``.

Service Handles

Whenever a listening partner initializes a service by executing `adap.Listen`, it receives as a result of that function an identifying integer known as a *service handle*. From the point of view of the application, this integer is simply an abstract identifier. In the case of `adap.Listen`, the service handle identifies the listening partner to itself.

When the listening partner accepts a request to connect from a connecting partner, a service handle identifying that partner is generated. Similarly, when a connecting partner requests a connection with a listening partner, a service handle identifying the listener is generated. These service handles should be saved because they are means by which partners are identified.

There are four types of service handles, including the three just mentioned:

- a *listen handle* identifies the listening partner to itself, typically for the purpose of closing the communication channel at some later time, and is the result of `adap.Listen`;
- an *accept handle* identifies a connecting partner to the listening partner, and is the first argument to the listening partner's callback function when a connected event occurs;
- a *connection handle* identifies the listening partner to a connecting partner, and is the first argument to the connecting partner's callback function when a connected event occurs;
- a *timer handle* identifies a timer, and is the result of `adap.SetTimer`.

Every service handle, including those for timers, should be saved for the purpose of closing the connection later.

Timeouts in Synchronous Communication

When a process sends a message to its partner synchronously, if no provision had been provided to interrupt the sending function, it will not return until the message is sent. If time is crucial, the wait may be unacceptable; it might be better to interrupt the sending function and try again later. The function that adap provides for sending messages synchronously, `adap.Syncsend`, has a timeout argument. If the timeout expires before the message is completely sent, the underlying message-sending mechanism is interrupted and `adap.Syncsend` returns. There is a timeout argument to the function for synchronously reading messages as well.

If a timeout occurs then all that is known about the message is that it has been partially sent or partially received, but the exact state is unknown. The only way to clear the pending message fragments is to force a reset.

A timeout argument indicates the duration of time during which attempts will be made to complete a synchronous operation. The form of a timeout is either a numeric scalar or one-element vector, or a two-element integer vector. In the case of a numeric scalar or one-element vector, the value represents seconds, and any fractional part of the number represents fractions of seconds. In the case of a two-element integer vector, the first element represents seconds and the second element represents microseconds. For example, 2, 2.5, and 2 500000 are all valid timeouts.

An alternative timeout form is to specify the clock time by which a synchronous operation must be completed, or else abandoned. This form is a three-element integer array, where the first element represents seconds since the Epoch (see `sys.secs_in_Epoch`), the second element represents microseconds, and the third element is 1.

The two forms of timeouts are called *duration timeouts* and *clock timeouts*.

Communication Errors

In asynchronous operations, an unrecoverable system error results in a callback with event type `` `error ``, while a recoverable system error results in a callback with event type `` `reset ``. For synchronous operations, however, no analogous default actions are taken. This means that the application writer is responsible for examining the error and determining if a `` `reset `` needs to be sent to the partner (see `adap.Reset`).

The errors that the synchronous functions can report (as indicated by the second element of an error return) are listed in the next table.

Errors in Synchronous Communication

Error Name	Description And Action To Be Taken
`` `buffread ``	Call `adap.Reset`.
`` `buffwrite ``	Call `adap.Reset`.
`` `fdsisset ``	Call `adap.Reset`.
`` `nochan ``	Call `adap.Reset`.
`` `select ``	Call `adap.Reset`.
`` `readImport ``	This error indicates that `adap.Syncread` received a bad A+ object. It may or may not require a reset. More likely, it indicates a bug in the partner. (A bad A+ object is one that does not conform to the A protocol, which must be used in synchronous communication; see "Protocols for Sending and Receiving Data".)
`` `export ``	This error indicates that the argument to `adap.Syncsend` is an invalid A+ object (perhaps one with function pointers). Reset is not necessary.
`` `interrupt ``	This indicates that a system interrupt occurred while the operation was in progress. A reset need not be called, but you should continue execution so that the interrupt can be processed. (Note that processing the interrupt may well cause a reset or even termination of the program.)
`` `timeout ``	This indicates that the function did not finish before the timeout was reached. Whether or not a reset is necessary is up to the application.

Definitions of adap Functions and External Functions

Asynchronous Send `adap.Send{h;x}`

Arguments and Result

`h` is a service handle, as described in "Service Handles". `x` is any array. The result is a scalar integer.

Definition

This function is used by either communicating partner to send a message to the other one. The message is the array *x*. If possible, the message is sent immediately. If not, the message is considered pending, and will be sent at some later time. *adap.Send* sends only one packet (2K, perhaps) synchronously (i.e., before it returns); the rest is transmitted during later calls to this function or when the process returns to the mainloop. If the message is sent immediately, any pending messages will also have been sent. The contents of the message *x* depend on the protocol being used (see "Protocols for Sending and Receiving Data").

The result is the number of messages sent. The number is 0 if the current message *x* is not immediately sent, and it may be greater than 1 if this message and pending ones are immediately sent. Any messages immediately sent by this function will not cause a *`sent* event; see "Protocols for Sending and Receiving Data". Note that a result of 0 can also arise if the function fails, which can happen, for example, if the arguments are invalid.

A cumulative total of the results of this function and the *call_data* arguments to *`sent* callbacks equals the total number of messages sent to the partner.

Close Handle *adap.Close{h}*

Argument and Result

h is a service handle, as described in "Service Handles". The result is a scalar integer.

Definition

This function closes the service handle *h*. All system resources associated with this service handle are deleted. In the case the service handle *h* was established through *adap.Listen*, previously established connections for the listening partner are maintained, but no new ones will be established. If the listening partner wishes to terminate service completely, in addition to closing its service handle *h*, it must explicitly close all accept handles it has previously received for *h*; the listening partner is responsible for maintaining a list of accept handles.

Service handles for timers can be closed with *adap.Close* before their set times expire; they are automatically closed when their set times expire, and therefore, the timers can be said to expire.

The result of this function is 1 if the connection is successfully closed, i.e. if the service associated with it is found and closed, and 0 otherwise.

The requirement to explicitly close connections when they are no longer needed is the main reason for saving the service handles produced by *adap.Connect*, *adap.Listen*, and *adap.SetTimer*.

Connect *adap.Connect{s;f}*

Arguments and Result

s is a service descriptor, as described in "Service Descriptors for Interprocess Connections". *f* is a callback function, as described in "Callback Functions and Events". The result is a service handle, as described in "Service Handles".

Definition

This function is used by a connecting partner to establish communication with a listening partner. See "Interprocess Communication". The result is -1 if the function fails, which can happen, for example, if either argument is invalid.

Debug Flag `adap.Debug{f}`

Argument

The argument is a scalar integer.

Definition

Immediately after execution of `adap.Debug{1}`, adap displays trace messages in the A+ session as it is executed. Immediately after execution of `adap.Debug{0}`, most adap messages are suppressed; some error and warning messages are still issued. A "zero-length message" warning indicates a real problem with a socket, and, even though the message may be suppressed, the condition may cause the process to burn CPU cycles.

When a problem occurs with adap, you can help the adap developers resolve it if you can repeat the error after executing `adap.Debug{1}` and send them the resulting A+ session log.

Export Data `adap.Export{x}`

This function has been superseded by `sys.exp`.

Get Attribute Value `adap.Of{h;s}`

Arguments and Result

The left argument `h` is a service handle, as described in "Service Handles". The right argument `s` is a symbol. The result is an array.

Definition

The result is the value of the attribute named in the right argument `s` for the service handle `h`. See `adap.Has`.

Get Client Data `adap.GetClientData{h}`

Argument and Result

The argument is a service handle, as described in "Service Handles". The result is an array.

Definition

The result of this function is the client data previously associated with `h` by `adap.SetClientData`. If no client data has previously been set or if `h` is not a currently active service handle, the value is Null.

Get Port and Workstation Information `adap.GetPort{h}`

Argument and Result

The argument is a service handle, as described in "Service Handles". The result is a five-element integer vector.

Definition

The first element of the result is the port number for the service handle, and the last four elements are the conventional Unix network id for the current host machine.

Get Timeout `adap.GetTimeout{t}`

Arguments and Results

The argument `t` is a duration timeout, as described in "Timeouts in Synchronous Communication". The result is a clock timeout.

Definition

The argument `t` represents a number of seconds, possibly fractional. The result is the clock time `t` seconds after the function is called. For example, `adap.GetTimeout{10.5}` is the clock time 10.5 seconds from the time the function was called.

This function is useful when several synchronous operations are to be performed in a row, and all must finish within a certain time. Since the operations are performed sequentially, the same duration timeout cannot be used for all calls, but the same clock timeout can. This function converts a duration timeout to the (nearly) equivalent clock timeout, if the first of the synchronous operations is called immediately after the conversion is done.

Import Data `adap.Import{x}`

This function has been superseded by `sys.imp`.

Listen `adap.Listen{s;f}`

Arguments and Result

`s` is a service descriptor, as described in "Service Descriptors for Interprocess Connections". `f` is a callback function, as described in "Callback Functions and Events". The result is a service handle, as described in "Service Handles".

Definition

This function is used to initialize listening partners, i.e., to set up the mechanism by which processes listen for partners attempting to establish connections. See "Interprocess Communication". The result is -1 if the function fails, which can happen, for example, if either argument is invalid. See "Service Descriptor Attributes regarding retries.

The listening partner can set the port number to 0 in the argument s, in which case adap generates a valid port number. The listening partner can then get the assigned port number by executing `adap.GetPort`, but must broadcast this number to all potential partners.

Example

```
(ListenHandle)←adap.Listen{
                            (`name;`Test;
                             `host;`localhost;
                             `port;0;
                             `protocol;`A;
                             `retry;0);
                            CallBackFunc};
```

Modify Timer `adap.ModifyTimer{h;s;d}`

Arguments and Result

`h` is the service handle of the timer, as described in "Service Handles". `s` is a symbol scalar, the name of the timer. `d` is a numeric scalar, the duration of the timer. The result is an integer scalar.

Definition

This function changes the name (`s`) or the duration (`d`) of the timer with service handle `h`. The new duration is from the time when `adap.ModifyTimer` was called. The result is 1 if the timer is found and successfully modified, and 0 otherwise.

Reset `adap.Reset{h}`

Argument and Result

`h` is a service handle, as described in "Service Handles". The result is a scalar integer.

Definition

If `h` is the service handle of an interprocess connection, this function closes the connection. This is a user-generated reset. It differs from a normal reset, in that a reset event is not generated. However, if this function is called by a connecting partner, the underlying system will then continually try to reconnect to the partner, just as it would for a normal reset. This function, which in practice would rarely be executed by the listening partner, has the same effect as `adap.Close` for that partner.

If `h` is a timer, the timer event is reset to its original time duration, and no callback is generated.

The result of this function is 1 if the service is successfully reset, i.e. if the service associated with it is found and reset, and 0 otherwise.

Set Attribute Value `adap.Has{h;x}`

Arguments

The left argument h is a service handle, as described in "Service Handles". The right argument is a nested vector.

Definition

The right argument is an association list, i.e., a nested vector consisting of symbol, value pairs. The symbols are the names of service handle attributes. The effect of this function is to set these attributes, for the service handle `h`, to the values in the symbol, value pairs. The attributes

applicable to `h` depend on the protocol for sending and receiving data with which `h` was established. A description of all the attributes follows.

- list: A list of names, as symbols, of the attributes applicable to the service handle `h`. This attribute applies to all protocols.
- rBufsize: The read buffer size for the A protocol; the default is 32K.
- rEventMode: For the A protocol. If `` `single `` (the default), a separate read event occurs for each message received. If `` `burst ``, all pending messages from a service handle are gathered in a nested array (even if there is only one) for a single read callback. (`Syncread` is not affected; it always returns one message and does not enclose it in a nested array.) The purpose of burst mode is to allow an application to ensure that it is up to date. Burst mode can prevent a process with time-consuming read-event code from falling behind and it can obviate the execution of time-consuming dependency definitions. If an application nevertheless falls behind, that fact can be seen from the increasing numbers of messages at each read event. Cf. the read event type.
- rPause: If 1, messages coming in on the read channel of the service handle `h` will be queued and no `` `sent `` events will be received; if 0, message processing proceeds normally, and in particular, any messages queued while the value was 1 will be read. This attribute applies to all protocols. Contact the A+ Development Group before setting this attribute.
- rPriority: The value is an integer representing the relative priority of processing the read channel of the service handle `h` in the event loop of the A+ process. (Every time through the event loop, events associated with the event source of highest priority are processed first, then those associated with the event source of next highest priority, and so on.) Note: screen management events have priority 1024. Contact the A+ Development Group before setting this attribute.
- rQueueStatus: This attribute is reference only, and its value and meaning are the same as those of the result of `adap.ReadQueueStatus{h}`. This attribute applies to all protocols.
- wPause: Like rPause, except that when 1, sent messages are queued. This attribute applies to the A, raw, and ipc protocols.
- wPriority: The value is an integer representing the relative priority of processing the send channel of the service handle `h` in the event loop of the A+ process. See rPriority. This attribute applies to the A, raw, and ipc protocols.
- wQueueStatus: This attribute is reference only, and its value and meaning are the same as those of the result of `adap.WriteQueueStatus{h}`. This attribute applies to the A, raw, and ipc protocols.
- wBufsize: The write buffer size for the A protocol; the default is 32K.
- wNodelay: For the A protocol. Causes even tiny messages to be sent immediately; prevents their being accumulated with a view toward traffic reduction by batch transmission.

Set Client Data `adap.SetClientData{h;x}`

Arguments and Result

`h` is a service handle, as described in "Service Handles". `x` is any array that satisfies the conditions of the A protocol (see "Protocols for Sending and Receiving Data"). The result is 0.

Definition

This function associates the array *x* with the service handle *h*. The array can be retrieved using *adap.GetClientData*. The client data *x* will no longer be accessible once the connection is closed.

Client data is a convenient way to store auxiliary information about the communication handle. For example, if a client has several communication paths open to the same server, the client data could hold "instance" information.

Set Timer *adap.SetTimer{h;f;d}*

Arguments and Result

h is a symbol scalar, which is the name of the timer and of the event type in the callback. *f* is a callback function, as described in "Callback Functions and Events". *d* is a numeric scalar that represents the duration of the timer, in seconds; *d* can be fractional. The result is a service handle, as described in "Service Handles".

Definition

This function sets the timer named by *h* to the duration specified by *d*. See "Interprocess Communication". The result is 0 if the function fails to establish a timer, which can happen, for example, if any of the arguments are invalid.

Status of the Read Queue *adap.ReadQueueStatus{h}*

Arguments and Result

The argument *h* is a service handle, as described in "Service Handles". The result is a two-element integer vector.

Definition

The first element of the result is 1 if there is a message waiting to be read, and 0 otherwise. The second element is 1 if there is a message in the process of being read.

Status of the Write Queue *adap.WriteQueueStatus{h}*

Arguments and Result

The argument *h* is a service handle, as described in "Service Handles". The result is a two-element integer vector.

Definition

The first element of the result contains the number of pending messages on the write queue. The second element is 1 if there is a message in the process of being sent, and 0 otherwise.

If the write queue is empty (no pending messages), the result is 0 0. A result of 2 0 indicates two pending messages, while 1 1 indicates one pending message which has been partially sent.

This function can be used in simple communication arrangements to keep track of the number of messages sent, rather than counting results of *adap.Send* and the callback data for sent events (see *adap.Send*, and the event types table).

Synchronous Exchange `adap.SyncXch{h;x;t}`

Arguments and Result

The argument `h` is a service handle, as described in "Service Handles". The argument `x` is the array to be the sent to the partner, which must satisfy the conditions of the A protocol (see "Protocols for Sending and Receiving Data"). The argument `t` is a timeout (see "Timeouts in Synchronous Communication"). The result is a nested vector.

Definition

`adap.SyncXch` is an A+ cover function for synchronous communication in which the message `x` is sent with `adap.Syncsend` and, in response, a message is received using `adap.Syncread`. This is useful when only one message will be received for each message sent. If more than one message will be received for each message sent, then `adap.Syncsend` should be used, followed by a series of calls to `adap.Syncread`. The latter occurs, for example, when a server performing a long computation sends a series of status reports to the client.

If `adap.SyncXch` fails due to a timeout in `adap.Syncread`, then the message was successfully sent, but the response was not received in the allotted time. If this function is subsequently called again for the same message, the message will be sent again.

The first element of the result is either `` `error `` or `` `OK ``. If `` `OK ``, the second element contains the message from the partner. If it is `` `error ``, the first three elements are the result from either `adap.Syncsend` or `adap.Syncread`, and the fourth element is either `` `send `` or `` `read ``, to indicate which function it is.

Synchronous Read `adap.Syncread{h;t}`

Arguments and Results

The argument `h` is a service handle, as described in "Service Handles". The argument `t` is a timeout (see "Timeouts in Synchronous Communication"). The result is a three-element nested vector, or the Null.

Definition

This functions waits for a message, i.e. array, to be received from the partner. The partner may send this array synchronously or asynchronously; if it was sent asynchronously, it must have been sent using the A protocol (see "Protocols for Sending and Receiving Data").

If `h` is bad (not a valid service) the result is null. Otherwise, the result is a three-element enclosed vector. The first element is either `` `error `` or `` `OK ``, indicating success or failure. In case of an error, the second element is a symbol categorizing the error, and the third element is a system-generated character vector describing the error in more detail. Otherwise, the second element is the received message, and the third element is the Null.

Synchronous Send `adap.Syncsend{h;x;t}`

Arguments and Result

The argument `h` is a service handle, as described in "Service Handles". The argument `x` is the array to be the sent to the partner, which must satisfy the conditions of the A protocol (see "Protocols for Sending and Receiving Data"). The argument `t` is a timeout (see "Timeouts in Synchronous Communication"). The result is a three-element nested vector, or the Null.

Definition

This function is for sending arrays to partners synchronously. This means that the message `x`, and any that are pending, are sent before the function terminates.

If `h` is bad (not a valid service) the result is null. Otherwise,the result is a three-element nested vector. The first element is either `` `error `` or `` `OK ``, indicating success or failure. In case of an error, the second element is a symbol categorizing the error, and the third element is a system-generated character vector describing the error in more detail. In case of success, the second argument is the numbers of messages actually sent, and the third argument is the status of the write queue (see `adap.WriteQueueStatus`). Note that if there were previously sent messages in the write queue when `adap.Syncsend` was called, then the following are possible:

- the number of messages sent can be greater than 1;
- if the timeout expires then the number of messages sent can be greater than 0, although the current message `x` was not sent.

7. The b Context

The b context is native to A+. It is not to be loaded.

Sorted Arguments to the b-Context Functions

The definitions of the b context functions refer to sorted arrays, or to permutation vectors that will reorder arrays in sorted order. If the array is simple, i.e., of depth 0, then sorted order means that the items are in nondescending, lexicographic order, which is described in the definition of Grade up. A permutation vector that puts a simple array in sorted order is therefore one that rearranges the items into nondescending, lexicographic order.

Nested scalars and vectors of depth 1 can also be in sorted order. In the case of a scalar `a`, this means that `>a` is in sorted order, while for a one-element vector `a`, it means that `0⊃a` is in sorted order. A nested vector `a` with two or more elements is said to be in sorted order if it is ordered like a dictionary with the items of `0⊃a` corresponding to the first letter of words, the items of `1⊃a` corresponding to the second letter, and so on. That is, the items of `0⊃a` are in sorted order; if `i` is any simple integer vector in increasing order for which the items of `i#0⊃a` are identical, then `i#1⊃a` is in sorted order; and so on.

For example, if

```
c←3 5⍴'blue blue green'
n←10 20 5
```

then `c` is sorted, `n` is not sorted, but the nested pair `(c;n)` is sorted.

Uniform Frames and Comparable Cells

Two simple arrays `a` and `b` are said to have *comparable `n`-cells* if their `n`-cells have the same shape, i.e., `(-n)↑⍴a` equals `(-n)↑⍴b`. If `a` has rank k then the items of `a` are (k-1)-cells. The array `b` is said to have *cells comparable to the items of* `a` if `a` and `b` have comparable (k-1) cells. For example, the right argument of the primitive function Find has cells comparable to the items of the left argument.

If `a` and `b` are nested scalars of depth 1 then they are said to have comparable n-cells if `>a` and `>b` have comparable n-cells, and analogously, `b` is said to have cells comparable to the items of `a` if this is true of `>a` and `>b`. If `a` and `b` are nested vectors of depth 1, then they are said to have comparable n-cells if for every valid scalar index `i` of both `a` and `b`, the simple arrays `i⊃a` and `i⊃b` have comparable n-cells, and analogously for `b` having cells comparable to the items of `a`. In particular, `a` and `b` do not have to be of the same length.

Suppose `a` and `b` are nested vectors and `b` has cells comparable to the items of `a`. Then for every valid scalar index `i`, if `r` is the rank of `i⊃a`, then the `(r-1)`-cells of `i⊃b` have the same shape as the items of `i⊃a`. Note that `r` may vary with `i`. The complementary frame of `b` has shape `(-r-1)↓⍴b`. If all the complementary frames for all `i` are equal, `b` is said to have *uniform complementary frames*. Similarly, if all the counts `#i⊃a` are the same for all `i`, then `a` is said to have *uniform counts*.

For example, if

```
c←3 5⍴'blue greengreen'
n←10 20 15
d←2 2 5⍴'greenblue red  green'
m←2 2⍴5 10 25 15
```

then the nested pair `(d;m)` has cells comparable to the items of the nested pair `(c;n)` and uniform complementary frames (the 2-frames of `d` and `m`), while `(c;n)` has uniform counts.

Definitions of b-Context Functions

Binary End `b.e{y;x}`

Arguments and Result

See Binary Iota, `b.i{y;x}`.

Dependency

Comparison Tolerance.

Definition

`b.e` differs from `b.i` in only one way: the elements of the result are last occurrences instead of first. That is, each element `n` in the result corresponds to a cell or cross section of `x`, and the value `n` is the largest index for which the `n`th item or cross section of `y` is identical to the

corresponding cell or cross section of `x`. If there is no such item or cross section of `y`, then `n` is the number of items (`#y`) or cross sections (`#0⊃,y`) of `y`.

Binary End, with Permutation `b.pe{y;p;x}`

Arguments and Result

See Binary Iota, with Permutation, `b.p{y;p;x}`.

Dependency

Comparison Tolerance.

Definition

`b.pe{y;p;x}` is to `b.e{y;x}` as `b.p{y;p;x}` is to `b.i{y;x}`. See Binary End; Binary Iota, with Permutation; and Binary Iota.

Binary Greater than or Equal to `b.ge{y;x}`

Arguments and Result

See Binary Iota, `b.i{y;x}`.

Dependency

Comparison Tolerance.

Definition

The definition of `b.i{y;x}` is based on comparing whether or not subarrays or cross sections of `x` and `y` are identical, depending on whether `x` and `y` are simple or nested arrays. The definition of `b.ge{y;x}` is exactly the same, except that the comparison function is not "identical to", but "greater than or equal to". That is, in comparing two simple subarrays `b` and `a` of `y` and `x`, respectively, `a` is said to be greater than or equal to `b` if `a` does not come before `b` in lexicographic order. In the case where `a` and `b` are cross sections of the nested arrays `x` and `y`, respectively, `a` is said to be greater than or equal to `b` if for every valid scalar index `j`, the simple array `j⊃,a` is greater than or equal to the simple array `j⊃,b` (the ravels of `a` and `b` are used to include the case where they are scalars).

Note that when `x` and `y` are simple, the definition of `b.ge{y;x}` is identical to the definition of the primitive Bins function, `y⍋x`, except that for `b.ge`, comparison tolerance is used when `x` and `y` are floating-point arrays.

Binary Greater than or Equal to, with Permutation `b.pge{y;p;x}`

Arguments and Result

See Binary Iota, with Permutation, `b.p{y;p;x}`.

Dependency

Comparison Tolerance.

Definition

`b.pge{y;p;x}` is to `b.ge{y;x}` as `b.p{y;p;x}` is to `b.i{y;x}`. See Binary Greater than or Equal to; Binary Iota, with Permutation; and Binary Iota.

Binary Iota `b.i{y;x}`

Arguments and Result

Both arguments `x` and `y` must be of the same type, one of character, integer, floating point, and boxed. The result is an integer array. If both are nested, then they must be scalars or vectors of depth 1.

In addition, `x` must have cells comparable to the items of `y`. If `x` and `y` are nested then `x` must have uniform complementary frames, `y` must be of uniform count (see "Uniform Frames and Comparable Cells"), and `y` must be sorted (see "Sorted Arguments to the b-Context Functions"). The shape of the result is the common value of the shapes of the complementary frames of `x`.

Dependency

Comparison Tolerance.

Definition

When `x` and `y` are simple arrays, the definition of `b.i{y;x}` is identical to the definition of Find, `y⍳x`, except that in the case of `b.i`, the items of the left argument `y` must be in nondescending lexicographic order.

When `y` is a nested array, its contents are best viewed as fields in a database. Let `j` be any valid index of both `,x` and `,y`. From the database viewpoint, the disclosed element `j⊃,y` is a field and its items `(j⊃,y)[k]` are records in that field. Cross sections of records `k⌷¨,y` can be formed for scalar indices `k`. Since all fields `j⊃,y` have the same number of items, every record is in some cross section.

According to the requirements on nested `x` and `y`, the cells in `j⊃,x` of rank `0⌈(⍴⍴j⊃,y)-1` can also be viewed as records in the field `j⊃,y`. The corresponding frames for all `j⊃,x` have the same shape, and so cross sections of records `(<⍳)⌷¨,x` can also be formed for `x`.

For every cross section `(<⍳)⌷¨,x` there is a corresponding element `n` of the result, which is the smallest index for which that cross section is identical to the cross section `(⌷x)↑n⌷¨,y`. If there is no such index, then `n` is the number of cross sections in `y`, i.e., `0⊃,y`.

Example

```
    c←3 5⍴'blue greengreen'
    n←15 10 20
    y←(c;n)
    d←2 2 5⍴'greenblue red  green'
    m←2 2⍴5 10 25 15
    x←(d;m)
    b.i{c;d}
1 0
```

```
3 1
    b.i{n;m}
3 3
2 1
```

In the last evaluation the left argument `n` is not sorted, and the result is not what we expect. The permutation `1 0 2` will put `n` in sort order. We can evaluate `b.i{1 0 2#n;m}`, or equivalently `b.p{n;1 0 2;m}`:

```
    b.p{n;1 0 2;m}
3 1
2 0
```

Even though `n` is not sorted, `(c;n)` is:

```
    b.i{y;x}
3 0
3 1
```

Binary Iota, with Permutation `b.p{y;p;x}`

Arguments and Result

The arguments `y` and `x`, and the result, are as described for Binary Iota, `b.i`. The argument `p` is a vector of nonnegative, nonrepeating integers.

Dependency

Comparison Tolerance.

Definition

The purpose of this function is to provide the advantages of `b.i` in the cases when a potential left argument to `b.i` is too large to reorder into nondescending order, as is the case with many mapped files.

If `x` and `y` are simple arrays, then the argument `p` is a vector of indices of the items of `y` such that the elements of `p#y` are in nondecreasing, lexicographic order. The number of elements in `p` is not necessarily equal to the number of items of `y`. The expression `b.p{y;p;x}` is formally equivalent to `b.i{p#y;x}`, but the permuted array `p#y` is never actually formed.

If `x` and `y` are nested arrays then for every valid index `j` of `,x` and `,y`, the argument `w` is a vector of indices of the items of `j⊃,y` such that the elements of `w#j⊃,y` are in nondecreasing, lexicographic order. `b.p{y;p;x;}` is formally equivalent to `b.i{z;x}`, where

```
z←y
z[⍳#x]←(<p)#¨z[⍳#x]
```

but `z` is never actually formed.

Binary Less than or Equal to `b.le{y;x}`

Arguments and Result

See Binary Iota, `b.i{y;x}`.

Dependency

Comparison Tolerance.

Definition

`b.le` differs from `b.ge` in only two ways: the comparisons are made with "less than or equal to" instead of "greater than or equal to", and the elements of the result are last occurrences instead of first.

That is, in comparing two simple subarrays `b` and `a` of `y` and `x`, respectively, `a` is said to be less than or equal to `b` if `a` does not come after `b` in lexicographic order. In the case where `a` and `b` are cross sections of the nested arrays `x` and `y`, respectively, `a` is said to be less than or equal to `b` if for each valid index `j`, the simple array `j⊃,a` is less than or equal to the simple array `j⊃,b` (the ravels of `a` and `b` are used to include the case where they are scalars).

Secondly, each element `n` in the result is the largest index for which the `n`th item or cross section of `y` is less than or equal to the corresponding cell or cross section of `x`. If there is no such item or cross section of `y`, then `n` is the number of items (i.e., `#y`) or cross sections of `y`.

Binary Less than or Equal to, with Permutation `b.ple{y;p;x}`

Arguments and Result

See Binary Iota, with Permutation, `b.p{y;p;x}`.

Dependency

Comparison Tolerance.

Definition

`b.ple{y;p;x}` is to `b.p{y;p;x}` as `b.le{y;x}` is to `b.i{y;x}`. See Binary Iota, with Permutation; Binary Less than or Equal to; and Binary Iota.

Binary Range `b.r{y;x}`

Arguments and Result

See Binary Iota, `b.i{y;x}`. The result is an integer array whose shape is `2,⍴b.i{y;x}`.

Dependency

Comparison Tolerance.

Definition

`0#b.r{y;x}` is identical to `b.i{y;x}`, and the elements of `1#b.r{y;x}` are the number of occurrences of the cells or cross sections of `x` in `y`.

Binary Range, with Permutation `b.pr{y;p;x}`

Arguments and Result

See Binary Iota, with Permutation, `b.p{y;p;x}`. The result is an integer array whose shape is `2,⍴b.p{y;x}`.

Dependency

Comparison Tolerance.

Definition

`b.pr{y;p;x}` is to `b.p{y;p;x}` as `b.r{y;x}` is to `b.i{y;x}`. See Binary Iota, with Permutation; Binary Range; and Binary Iota.

Binary Unique `b.u{y;x}`

Arguments and Result

The arguments are simple and the items of `y` are assumed to be unique. Otherwise, the arguments and result follow the rules for `b.i{y;x}` in the simple case.

Dependency

Comparison Tolerance.

Definition

The result is identical to Binary Iota, `b.i{y;x}`. Since, however, unique items in `y` are assumed, the algorithm is slightly faster. The use of this function is discouraged, as it may be removed.

8. The c Context

The c context is native to A+. It is not to be loaded.

The c context provides utilities for creating, manipulating, and examining C-language structures and pointers to data. Pointers are represented as scalar integers in A+. The representation of structures is more complicated, and is discussed below in "The Representation of C-Language Structures in A+". The definitions of the c-context functions for structures follow that section.

The Representation of C-Language Structures in A+

A C-language structure is represented in A+ by three vectors: a vector of symbols for member names that the user can choose; a vector of integers defining the length of each member; and a vector of symbols describing the storage type of each member. The symbols for specifying storage types are given in the table "A+ Symbols for Specifying C Data Types". The storage types `` `struct1 ``, `` `struct2 ``, `` `struct3 ``, `` `struct4 `` are for members that are structures, and in these cases the corresponding element of the length vector is the length of the structure, in bytes. In all other cases the element of the length vector is the number of elements in the

member; if the length is greater than 1, then the member is assumed to be an array, whereas if the length is 1 the element is assumed to be a scalar.

A C-language structure created from these vectors can be passed to C programs that have been dynamically loaded to the A+ session. Each C datatype is properly aligned in the structure for the machine on which A+ is running.

A+ Symbols for Specifying C Data Types

A+ Symbol	C Data Type		A+ Symbol	C Data Type
`` `char ``	char		`` `struct1 ``	char aligned on one-byte boundary
`` `u_char ``	unsigned char		`` `struct2 ``	char aligned on two-byte boundary
`` `short ``	short		`` `struct4 ``	char aligned on four-byte boundary
`` `u_short ``	unsigned short		`` `struct8 ``	char aligned on eight-byte boundary
`` `int ``	int		`` `pointer ``	void*
`` `u_int ``	unsigned int		`` `float ``	float
`` `long ``	long		`` `double ``	double
`` `u_long ``	unsigned long			

Functions that Modify Arguments

Functions in this chapter, e.g., `c.structset`, may modify one of their arguments. In such cases, the argument should have a name and the name should be used in the argument position, for then you can see the changes after the function has completed execution. In addition, you must be sure that other objects are not modified as well. The reason is that A+ manages reference counts on objects, meaning that at any point in an execution two or more names can actually share the same array value; sharing stops when the value of one of the objects changes in a normal A+ way. However, the modification of an array value by a function like `c.structset` is not a normal A+ value change, and all names that had previously shared the value will now shared the modified value, unless prevented. The remedy is to replace the named argument `a` with something like `a0←(⍴a)⍴a`.

Example

A simple example will serve to illustrate most of the functions in this chapter. Consider the C-language structures

```
struct s1 {
  int         fieldA;
  short       fieldB[4];
};

struct s2 {
  int         field1;
  struct s1   field2;
  struct s1   *field3;
```

```
};
```

The s1 structure is represented in A+ as:

```
s1_members←`fieldA`fieldB
s1_lengths←1 4
s1_types←`int`short
```

The function `c.structdef` produces another A+ representation that includes various information about the structure, such as offsets, and which is used to create the actual structures, insert values, and extract values:

```
s1←c.structdef{s1_members;s1_lengths;s1_types}
```

The second structure can now be represented:

```
s2_members←`field1`field2`field3
s2_lengths←1,(c.structsize s1),1
s2_types←`int,(c.structtype s1),`pointer
```

Structures with these definitions can now be realized:

```
s1_data←c.structcreate s1
s2←c.structdef{s2_members;s2_lengths;s2_types}
s2_data←c.structcreate s2
```

When you put data in the realized structures, the value returned by `c.structset` is the updated structure:

```
     c.structset{s1;s1_data;`fieldA`fieldB;(3;6 7 8 9)}
 3 393223 524297 0
     s1_data
 3 393223 524297 0
     c.structset{s2;s2_data;`field1;345}
 345 0 0 0 0 0
     c.structset{s2;s2_data;`field2`field3;
                 (s1_data;c.pointer{s1;s1_data})}
 345 3 393223 524297 67120480 0
```

You use `c.structget` to retrieve the data:

```
     c.structget{s1;s1_data;`fieldB}
 6 7 8 9
     m←c.structget{s2;s2_data;`field2}
     c.structget{s1;m;`fieldB}
c.structget{s1;m;`fieldB}
     p←c.structget{s2;s2_data;`field3}
     n←c.struct_pointed_to_by{s1;p}
     c.structget{s1;n;`fieldB}
c.structget{s1;m;`fieldB}
```

Display the contents of the structures:

```
      c.structprint{s1;s1_data}
fieldA:(int): 3
fieldB:(short): 6 7 8 9
     c.structprint{s2;s2_data}
field1:(int): 345
field2:(struct4): ----
field3:(pointer): 67120480
```

Definitions of c-Context Functions

A+ Array to Character Vector Representation `c.stuff{a}`

Argument and Result

The argument is an A+ array. The result is a character vector.

Definition

The result is a character vector representation of the of the A+ array `a`.

Note: The functions `sys.exp` and `sys.imp` should be used in place of `c.stuff` and `c.unstuff`.

A+ Array Header `c.AHeader{a}`

Argument and Result

The argument is any A+ array. The result is a five-element nested vector.

Definition

The result contains all the information in the internal header of an A+ array. The meaning of the elements of the result are in the tables "The Result of the Function c.AHeader" and "A+ Types vs. the Type Specification in c.AHeader Result".

The Result of the Function c.AHeader

Element Index	Description
0	An enclosed integer scalar holding the reference count of the array.
1	An enclosed character vector representing the type of the array (see next table).
2	An enclosed integer scalar holding the rank of the array.
3	An enclosed integer scalar holding the element count of the array.
4	An enclosed integer vector holding the shape of the array.

A+ Types vs. the Type Specification in c.AHeader Result

A+ Type	c.AHeader Value		A+ Type	c.AHeader Value
integer	`"It"`		derived function	`"Xt"`
floating point	`"Ft"`		user defined function	`"Xt+1"`

character	`"Ct"`		monadic operator	`"Xt+2"`
nested, symbol, or function array	`"Et"`		dyadic operator	`"Xt+3"`

Character Value at Pointer Location `c.char_pointed_to_by{i}`

Argument and Result

The argument is a one-element integer array. The result is a character scalar.

Definition

The integer `i` is a pointer to a C-language char value. The result is that value as an A+ scalar of type `` `char ``.

Character Vector at Pointer Location `c.string_pointed_to_by{i}`

Argument and Result

The argument is a one-element integer array. The result is a character vector.

Definition

The integer `i` is a pointer to a C-language char string. The result is that value as an A+ character vector.

Character Vector Representation to A+ Array `c.unstuff{a}`

Argument and Result

The argument is an A+ character vector. The result is an A+ array.

Definition

`c.unstuff` is a left inverse of `c.stuff`:

```
c.unstuff c.stuff x
```

is identical to `x` for any A+ array `x`.

Note: The functions `sys.exp` and `sys.imp` should be used in place of `c.stuff` and `c.unstuff`.

Define a Structure `c.structdef{f;l;t}`

Arguments and Result

The arguments `f` and `t` are vectors of symbols, while `l` is a vector of integers. All three vectors have the same length. The result is a general array of the form `(g;m;u;o;c)`, where `g` is identical to the argument `f`, `m` is identical to the argument `l`, `u` is identical to the argument `t`, and `o` and `c` are integer vectors of length one more than the length of the argument vectors.

Definition

The arguments `f`, `t`, and `l` represent a C-language structure (see "The Representation of C-Language Structures in A+"). For each integer `i` that is a valid index of the argument vectors `f`, `l`, and `t`, the triple `f[i]`, `l[i]`, and `t[i]` represents a member of the structure. The integer `o[i]` is the offset in bytes from the beginning of the structure to the beginning of this member. The integer `c[i]` is a type code representing the storage type of the member. If `n` is the number of elements in the argument vectors then `o[n]` and `c[n]` are both defined; `o[n]` is the size of the structure in bytes, and `c[n]` is the alignment factor for the structure.

The contents of `o` will be different for different computer architectures, so it is best not to store results of `c.structdef` in A+ script files.

Display the Contents of a Structure `c.structprint{s;a}`

Arguments

The left argument `s` is a five-element nested vector and the right argument `a` is an integer vector.

Definition

The argument `s` is a structure definition, i.e., a result of `c.structdef`. The argument `a` is a structure specified by `s`; at one point it was a result of `c.structcreate`, and most likely `c.structset`. The effect of this function is to display the contents of the structure `a`. Members that are structures are noted, but their values are not displayed.

Floating-Point Value at Pointer Location `c.float_pointed_to_by{i}`

Argument and Result

The argument is a one-element integer array. The result is a floating-point scalar.

Definition

The integer `i` is a pointer to a C-language float value. The result is that value as an A+ scalar of type `` `float ``.

Form `c.form`

Definition

This function used to be in the d context; d is a relational data base toolkit in A, the predecessor of A+. The function is provided in A+ for migration purposes.

Get Values from a Structure `c.structget{s;a;f}`

Arguments and Result

The argument `s` is a five-element nested vector and `a` is an integer vector. The argument `f` is either a one-element symbol array or a vector of symbols. If `f` has one element then the result is either a simple scalar or a simple vector with two or more elements; if `f` has more than one element, the result is a nested vector with the same number of elements as `f`, and each element of the result is either a simple scalar or a simple vector with two or more elements.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The argument *a* is a structure specified by *s*; at one point it was a result of *c.structcreate*, and most likely *c.structset*. The result holds the values in the structure *a* for the members specified by *f*.

Integer Value at Pointer Location *c.int_pointed_to_by{i}*

Argument and Result

The argument is a one-element integer array. The result is an integer scalar.

Definition

The integer *i* is a pointer to a C-language int value. The result is that value as an A+ scalar of type *`int*.

Pointer to an A+ Array Value *c.ptr{a}*

Argument and Result

The argument is any A+ array. The result is a scalar integer.

Definition

The result is a pointer to the data area of the A+ array *a*.

Pointer to a Structure Value *c.pointer{s;a}*

Arguments and Result

The left argument *s* is a five-element nested vector and the right argument *a* is an integer vector. The result is a scalar integer.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The argument *a* is a structure specified by *s*; at one point it was a result of *c.structcreate*, and most likely *c.structset*. The result is a pointer to the beginning of the actual data of the structure (in the data area of *a*).

Realize a Structure *c.structcreate{s}*

Argument and Result

The argument *s* is a five-element nested vector. The result is an integer vector.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The result is an integer vector large enough to hold a structure defined by *s*, and initialized to 0. The result is called a structure initialization.

Set Values in a Structure *c.structset{s;a;f;v}*

Arguments and Result

The argument *s* is a five-element nested vector and *a* is an integer vector. The argument *f* is either a one- element symbol array or a vector of symbols. If *f* has one element, then the argument *v* is a simple array. If *f* has more than one element, then *v* is a nested array of depth 1 with the same number of elements as *f*. This function directly modifies the value of the argument *a* (see "Functions that Modify Arguments"). The result is an integer vector.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The argument *a* is a structure specified by *s*; at one point it was a result of *c.structcreate*. The argument *f* specifies members of the structure *a*, and *v* specifies values for those members. The effect of this function is to assign these values to those members of *a*. A value in *v* can have a different number of elements than is defined for that member. If it has more elements, only as many as needed are inserted in *a*. If it has fewer, then the contents of the member beyond the values supplied by *v* may be unpredictable.

All values in *v* that correspond to integer or floating-point members in the structure should be of A+ type *`int* or *`float*, respectively. Values in *v* that correspond to structure members should be results of *c.structset* for those structures. The result of this function is the modified argument *a*.

Size of a Structure *c.structsize{s}*

Argument and Result

The argument *s* is a five-element nested vector. The result is an integer scalar.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The result is the size, in bytes, of the structure defined by *s*.

Similarly to *c.structdef*, the result of c.structsize will be different for different computer architectures, so it is best not to store these results in A+ script files.

Store a Character Value *c.place_chars_at{a;i}*

Arguments

The left argument *a* is a character array. The right argument *i* is a scalar integer.

Definition

The integer *i* is a pointer to a list of C-language char values. The effect of this function is put the elements of *a* in that list (in *,a* order).

Store a Floating-Point Value *c.place_floats_at{a;i}*

Arguments

The left argument *a* is a floating-point array. The right argument *i* is a scalar integer.

Definition

The integer *i* is a pointer to a list of C-language double values. The effect of this function is put the elements of *a* in that list (in , *a* order).

Store an Integer Value *c.place_ints_at{a;i}*

Arguments

The left argument *a* is an integer array. The right argument *i* is a scalar integer.

Definition

The integer *i* is a pointer to a list of C-language int values. The effect of this function is put the elements of *a* in that list (in , *a* order).

Structure Value at Pointer Location *c.struct_pointed_to_by{s;i}*

Arguments and Result

The left argument *s* is a five-element nested vector. The right argument *i* is a one-element integer array. The result is an integer vector.

Definition

The argument *s* is a structure definition, i.e., a result of *c.structdef*. The integer *i* is a pointer to a C-language structure that is equivalent to one defined by *s*. The result is an integer vector holding a copy of that structure.

Type Double Value at Pointer Location *c.double_pointed_to_by{i}*

Argument and Result

The argument is a one-element integer array. The result is a floating-point scalar.

Definition

The integer *i* is a pointer to a C-language double value. The result is that value as an A+ scalar of type *`float*.

Type Short Value at Pointer Location *c.short_pointed_to_by{i}*

Argument and Result

The argument is a one-element integer array. The result is a integer scalar.

Definition

The integer *i* is a pointer to a C-language short value. The result is that value as an A+ scalar of type *`int*.

Type of a Structure *c.structtype{s}*

Arguments and Result

The argument *s* is a five-element nested vector. The result is a symbol scalar or the integer 0.

Definition

The argument `s` is a structure definition, i.e., a result of `c.structdef`. The result is the type of the structure, i.e., one of `` `struct1 ``, `` `struct2 ``, `` `struct4 ``, or `` `struct8 ``, depending on whether the structure definition contains at most one-byte members, two-byte, four-byte, or eight-byte, respectively. Otherwise the result is 0, which would most likely indicate a damaged structure definition.

9. The p Context

Overview of Packages

The p context provides functions to support A+ package storage.

A *package* is a single structure containing A+ functions, variables, and dependencies. When a package is stored in a file, it is called a *packfile*. A *packstring* is a package in the form of a character array in an A+ workspace. A package can be used for storing a miscellaneous collection of A+ objects in a file, loading an application - it is usually faster than a script -, sending A+ objects through an adap connection, and performing lexical analysis (by a very knowledgeable person).

Packages are manipulated by external functions stored in the p context, which does not need to be loaded. The functions come in pairs: those whose unqualified names begin with `s` work on packstrings and those whose names begin with `f` work on packfiles. Package storage is optimized for fast retrieval, at the expense of space where necessary.

A package consists of a number of items, each of which is an object of one of these kinds:

- an A+ data object (possibly an enclosed primitive function or operator);
- a dependency - whose definition is included but not value, so that the dependency will be evaluated afresh when needed after retrieval;
- a defined function or operator;
- a definition of a mapped file - viz., the name of the file and an integer indicating the type of mapping: the data itself is not included in the package.

Nothing else can be included in a package. In particular, an item cannot be an s attribute on a variable, the state of a variable's display, an attribute set with `_set`, a set or preset callback, an external function (i.e., a function loaded using `_dyld`), or an external state, such as an adap connection.

The only operations performed during retrieval from a package are:

- specification of global variables;
- definition of dependencies;
- definition of functions and operators;

- Mapping (beaming) in of files, using `I` - note that a file's beamed value may be different from its value when it was put into the package and that there will be a value error if the file was removed in the interim.

If some other operation is required in order to restore some information, then you must provide a mechanism supplemental to package retrieval. For example, you could include in a package an initialization routine that sets attributes on variables.

Some of the restrictions, such as the one on dynamically loaded external functions, are due to an inability to retrieve the information needed to adequately store the object; they may possibly be lifted in the course of further development of the interpreter. Some objects, such as adap connections, are so complex that full retrieval of them may never be possible.

Other restrictions reflect a need for experience with packages and further thought. Callbacks, for instance, are not included because it is not clear whether during retrieval of a package variables should be specified and then callbacks set (so that no callbacks fire), callbacks should be set and then variables specified (so that all callbacks fire), or objects should be established in their order in the package definition (so that their order determines which callbacks fire).

Creation and Modification of Packages

To create a packstring or packfile, enter the appropriate one of these statements:

```
ps←p.snew{obj}

p.fnew{filenm; obj}
```

- *filenm* names a file. The name given in *filenm* will have `.pkg` appended to it if it does not already contain a period. *filenm* is one of:
  - a symbol;
  - a character vector;
  - a two-element nested array `(`*flnm*`;`*opts*`)` where *flnm* is a file name in one of the previous two forms and *opts* is a character string selecting options. Options are discussed in "Inquiry and Options", below.

- *obj* is either:
  - the names of the global objects to be stored in the package, as a symbol vector or scalar - unqualified names being taken to be in the root context, so `` `b `` and `` `.b `` are equivalent; or
  - a slotfiller, whose first element `(0⊃`*obj*`)` supplies the names of the objects to be stored in the package and whose second element `(1⊃`*obj*`)` supplies their values.

The result of `p.fnew` is null and that of `p.snew`, *ps*, is of course a packstring, which is a simple character string.

To add items to a package or replace them (where the names coincide), enter the appropriate one of these statements:

```
newps←p.sadd{ps; obj}
```

```
p.fadd{filenm; obj}
```

where the argument names have the same significance as in the previous pair of statements, except that *ps* can take the form of a nested vector `(p;opts)`, in which *p* is a packstring and *opts* is a character string selecting options, the same as the options for the corresponding form of *filenm*. The result of `p.fadd` is null, and *newps*, the result of `p.sadd`, is a packstring, which can be the one being modified, *ps*.

Retrieval of Objects and Information from Packages

To establish (fix) objects in the workspace from data and definitions in a package, enter the appropriate one of these statements:

```
p.sfix{ps; sel}

p.ffix{filenm; sel}
```

where *ps* and *filenm* are as in the previous pair of statements and *sel* is one of:

- Null - i.e., `p.sfix{ps;()}` or `p.sfix{ps;}` or similarly for `p.ffix` -, meaning that everything in the package is to be established in the workspace.
- A list of names to be fixed in the workspace, given as a scalar or vector of symbols.
- A two-element nested vector of the form `(pkgnames;wsnames)`, where the two elements are symbol scalars or vectors of the same length. The objects to be retrieved are listed in *pkgnames* and the names they are to be given in the workspace are listed in the corresponding positions in *wsnames*.

The explicit result of both functions is the Null.

The following two functions return data from a package in the form of a slotfiller, without establishing any global objects:

```
sf←p.sslot{ps; sel}

sf←p.fslot{filenm; sel}
```

where the arguments are as described above, except that *sel* is only null or a list of names, not a two-element nested vector. In the result, *sf*, the first element lists the names of the objects and the second gives their values; in other words, *sf* is in one of the allowable forms for the right argument in the creation and modification functions.

A function retrieved in this way is returned as a function scalar. Since there is no analogous form for a dependency, it cannot really be retrieved by these functions, and it has a null value in *sf*.

Inquiry and Options

A list of the items in a packstring or packfile can be obtained by entering the appropriate one of

```
p.scatalog{psname}

p.fcatalog{filenm}
```

They both produce symbol vectors.

An option string, which is a character vector, contains any of the following characters, with the indicated meanings:

d - debug: messages are to be generated by the package functions. Primarily for debugging.

h - hash table: the ...*new* and ...*add* functions are to make a hash table, for greater speed when only some items, not all, are being retrieved. Tables are retained in packages regardless of later settings of *h*. Primarily for packages used as component file systems.

p - pieces: ...*fix*, ...*slot*, and ...*catalog*, instead of operating normally, are to return the various parts of a package as a nested array. This option is primarily for debugging, but it also enables lexical analysis.

r - result: has the same effect as *p* except that ...*fix* is also to make assignments to global objects in the usual way.

s - stats: ...*new* and ...*add* are to issue messages containing statistics for the package just created.

v - verbose: the functions are to issue messages regarding objects handled, hash table creation, etc.

As indicated above, an option string can be included with an argument that names a file or string. When an option string is not so included, package functions use the default option string, whose initial value is `''`.

This default string can be changed by entering the statement

```
p.opts{optstring}
```

where *optstring* is the new default string.

Example

```
      b←10+c←27                  ⍝ Set up variables b and c
      f{x;y}:(|x)⌈|y             ⍝  and a function f
      d:f{b;c}                   ⍝  and a dependency d, and
      s←p.snew `b`c`d`f          ⍝  put them in a packstring s
      'test.m'⌶'abcdefghijklmnop'
      mf1←mf←1⌶'test.m'          ⍝ Create mapped file mf
      b←38; c←1000;              ⍝ Change values of b and c
      s←p.sadd{s;`b`mf}          ⍝  and change the b value in s
                                 ⍝  and put mapped file mf in s
      ⍝_ex¨ `b`c`d`f`mf
 0 0  0  0  0
      mf1[0 1 2]←'ABC'           ⍝ Change contents of file, using mf1

      _ex `mf1
 0
      p.sfix{s;}                 ⍝ Establish the objects in s
      d                          ⍝ Recall that we didn't
38                               ⍝  change c in s
      mf                         ⍝ mf has the latest value of the file.
ABCdefghijklmnop

      p.sslot{s;}                ⍝ Retrieve s as a slotfiller.
<   `.b `.c `.d `.f `.mf
```

```
< <  38
  <  27
  <                          ⍝ Null for the dependency.
  < .f                       ⍝ A function scalar for a function.
  < ABCdefghijklmnop         ⍝ The contents of the file.
      p.sslot{s;`b`c}        ⍝ Selective retrieval, as a
<  `.b `.c                   ⍝  slotfiller for visibility.
< <  38
  <  27

⍝ Now demonstrate a packfile, an option string,  and a slotfiller argument.


     p.fnew{('test.m';"vh");(`one`two`three;(3;2;1))}
⍝ PKG: 0: Storing .one       ⍝ The v elicits messages.
⍝ PKG: 1: Storing .two
⍝ PKG: 2: Storing .three
⍝ PKG: Storing hash table.   ⍝ The h creates a hash table.

     p.fslot{'test.m';`two`three}
<  `.two `.three             ⍝ Retrieved from the file.
< <  2
  <  1
```

10. The sys Context

Introduction

The sys context is native to A+. It is not to be loaded.

Many of the sys-context functions are equivalent to Unix system calls or C-library routines. These system calls and library routines are not described here, but only their relationships to the sys-context functions. See the system manual pages for documentation. For example, the documentation on the `chmod()` system call can be viewed by executing `man 2 chmod` in an XTerm session, while the documentation on the `times()` library routine can be viewed by executing `man 3 times`.

Functions that Modify Arguments

Some functions in this chapter, e.g., *sys.fcntl*, modify some of their arguments. In such cases, the argument should have a name and the name should be used in the argument position, for then you can see the changes after the function has completed execution. In addition, you must be sure that other objects are not modified as well. The reason is that A+ manages reference counts on objects, meaning that at any point in an execution two or more names can actually share the same array value; sharing stops when the value of one of the objects changes in a normal A+ way. However, when functions like *sys.fcntl* modify an array value, that is not a normal A+ value change, and all names that had previously shared the value will now share the modified value. The remedy is to replace the named argument *a* with something like *a*`0←(⍴a)⍴a`.

Path Names

Path names to files and directories are represented by character vectors. Since Unix file names can contain blanks, blanks in the character vectors are considered significant. In particular, trailing blanks are considered significant, and therefore should be removed from the character vector representations unless they are necessary. Both `sys.astat` and `sys.alstat`, however, do remove trailing blanks.

File Permissions Mode

When a Unix file is created, its access permission characteristics are set, specifying who can read, write, and execute the file. The access permission characteristics are represented numerically as 4 for read, 2 for write, and 1 for execute. In addition, any additive combination of these three is allowed. For example, a file that is both readable and executable has file access 4+1, or 5, while a file opened for both reading and writing is opened with access permission 4+2, or 6. All possible access permissions are represented by the integers 1 through 7.

Files have access characteristics for three independent classes of users: owner of the access, group access, and general access. If the access permissions are `o` for the owner, `g` for the group, and `u` for general users, then the access permissions of the file is the octal representation `8 8 8⊥o,g,u`.

Manifest Constants

Many of the sys-context functions are equivalent to Unix system calls. These system calls often have parameters which are specified by so-called manifest constants. For example, to open a file (in a C-language program) for reading only, one of the parameters to the open system call would be `O_RDONLY`. These constants can be specified in two ways in A+, either as symbols or integers. The symbol form is the most direct since it uses the names of the constants unchanged, and the names of these constants can be found in the "man page" documentation for the system calls. For example, in the case of the opening a file for reading only, the A+ function `sys.open` would be called with the second argument `` `O_RDONLY ``. The integer values of the constants can also be used, and can be found in the appropriate include files, although you are urged to use the symbols.

Warning! The integer values may differ for Sun, IBM, and other systems.

When more than one constant is to be specified, simply call the A+ function with the appropriate symbol vector. To use an integer argument in this case, OR the integer values of the individual constants together bitwise.

Definitions of sys-Context Functions

Export an Array `sys.export{a;t;f}`

Arguments and Result

The argument `a` is any A+ array that does not contain a function expression, `t` is a character vector of length 256 or 0 (or the Null), and `f` is an integer scalar. The result is a one- or two-element nested vector.

Definition

This function maps the object `a` to CDR (Common Data Representation) format for transferring among workstations or to APL systems. The argument `t` is a translation table. For example, if the object is to be sent to an APL2 system, then character (sub)arrays should be translated to EBCDIC character encoding. If the length of `t` is 0 then no translation occurs. The argument `f` is a boolean flag. If 1, symbols are not modified; if 0, which should be used when transferring A+ arrays to APL systems, symbols are converted to character vectors and the character translation defined by `t` is applied. If the mapping of `a` to CDR format is successful, the result is a two-element nested vector of the form `(0;v)`, where `v` is the CDR formatted object. If the mapping fails, the result is of the form `1⍴<n`, where the integer `n` is related to the length of the mapped portion of `a` at the time the error occurred.

The function `sys.exp` should be used in place of this function.

Import an A+ Array `sys.import{v;t}`

Arguments and Result

Both the left argument `v` and the right argument `t` are character vectors; the right argument can also be the Null. The length of the right argument is either 0 or 256. The result is a one- or two-element nested vector.

Definition

This function maps the CDR (Common Data Representation) formatted array `v` to an A+ object. The left argument `v` is the CDR formatted array. The right argument `t` is a translation table. For example, if the left argument `v` comes from an APL2 system, then character (sub)arrays must be translated from EBCDIC character encoding. If the length of `t` is 0 then no translation occurs. If `v` is successfully mapped to an A+ object then the result is a two-element nested vector of the form `(0;a)`, where `a` is the decoded A+ object. If `v` cannot be mapped then the result is `1⍴<1`.

The CDR format allows for a richer set of data types than that used by A+. These are listed in the table "CDR-to-A+ Type Conversion", along with the A+ data types to which they are mapped. The mapping fails if `v` contains elements of data types that are rejected.

The function `sys.imp` should be used in place of this function.

CDR-to-A+ Type Conversion

CDR Type	A+ Type
B1, B4 and B8 (1-bit, 4-bit, and 1-byte boolean)	`` `int ``
I2 and I4 (2-byte and 4-byte integer)	`` `int ``
C1 (1-byte character)	`` `char ``
E4 and E8 (4-byte and 8-byte floating point) (assumed to be in IEEE floating-point format)	`` `float ``
G0 (general array)	`` `box ``
S1 (A+ extension to the CDR format for symbols)	`` `sym ``

E16 (16-byte floating point)	rejected
J8, J16 and J32 are rejected (8-byte, 16-byte, and 32-byte complex numbers)	rejected
C4 (4-byte character)	rejected
A8 (integer progression)	rejected
Pn (packed decimal format)	rejected
Zn (zoned decimal format)	rejected
X0 (filler - the data in the object is meaningless)	rejected

Simplified Export `sys.exp{a}`

Argument and Result

The argument `a` is any A+ object that does not contain function expressions. The result is a character vector.

Definition

This function is a simplified version of `sys.export` for transferring A+ objects among workstations. That is, `sys.exp{a}` is essentially equivalent to `1⊃sys.export{a;();1)`. The result is the CDR format of `a`. If the translation fails, a domain error is signalled in the usual A+ manner. A lengthy call can be interrupted by a `SIGINT` (**Ctl-c Ctl-c** from an Emacs session). This capability allows one to rescue machines that are frozen because of a large `sys.exp` or `adap.Send` call.

Handles little-endian files and 64-bit platforms.

Simplified Import `sys.imp{v}`

Argument and Result

The argument `v` is a character vector. The result is an A+ array.

Definition

This function is a simplified version of `sys.import` for transferring A+ objects among workstations. That is, `sys.imp{a}` is essentially equivalent to `1⊃sys.import{a;()}`. The result is the A+ array represented by `v`. If the translation fails, a domain error is signalled in the usual A+ manner.

Handles little-endian files and 64-bit platforms.

Synchronize a Mapped File `sys.amsync{a;i}`

Arguments and Result

The left argument *a* is a mapped file, and the right argument *i* is an integer scalar or vector. The result is an integer scalar.

Definition

This function uses the `msync()` system call to synchronize the virtual memory pages of the mapped file with the file on disk. The value of the function is the value of that system call.

File to Character Matrix *sys.readmat{f}*

Arguments and Result

The argument *f* is a character vector. The result is a character matrix, or 0, or the Null.

Definition

The argument *f* holds a path name of a file (see "Path Names"). The result is a character matrix holding the contents of that file; blanks are appended to rows to make them the same length as the longest line in the file. If the function fails, the result is 0 or the Null.

exit *sys.exit{n}*

Argument and Result

Both the argument and result are integer scalars.

Definition

This function is equivalent to the `exit()` system call, for exiting from the process. The result of the function is the result of the system call. The A+ system function *_exit* should be used in place of this function.

kill *sys.kill{n;s}*

Arguments

The first argument is a scalar integer, the second a scalar symbol.

Definition

This function is equivalent to the `kill(pid, sig)` system call, for sending a signal (the second argument) to a process (the first argument). The signals are:

```
`SIGABRT  `SIGALRM   `SIGBUS   `SIGCHL
                               D
          `SIGCONT   `SIGEMT   `SIGFPE   `SIGHUP
          `SIGILL    `SIGINT   `SIGIO    `SIGIOT
          `SIGKILL   `SIGPIPE  `SIGPROF  `SIGQUIT
          `SIGSEGV   `SIGSTOP  `SIGSYS   `SIGTERM
```

`SIGTRAP `SIGTSTP `SIGTTIN `SIGTTOU

`SIGURG `SIGUSR1 `SIGUSR2 `SIGVTALRM

`SIGWINCH `SIGXCPU `SIGXFSZ

Flush Standard Out *sys.fflush_stdout{}*

Definition

This function is equivalent to the `fflush(stdout)` C-library call. The result is the Null.

getenv *sys.readenv{x}*

Argument and Result

The argument *x* is a character vector. The result is either a character vector or the Null.

Definition

This function is equivalent to the `getenv()` system call, which returns the value of the environment variable named in *x*. The result of the function is the result of the system call if the system call is successful, and the Null otherwise.

putenv *sys.setenv{x}*

Argument and Result

The argument *x* is a character vector. The result is an integer scalar.

Definition

This function is equivalent to the `putenv()` system call; the value of the argument *x* is of the form *"NAME=VALUE"*, and environment variable named in *NAME* is given the value specified by *VALUE*. The result of the function is the result of the system call.

sleep *sys.sleep{s}*

Argument and Result

The argument *s* is a numeric scalar and the result is a scalar integer.

Definition

This function suspends execution for *s* seconds. If *s* is an integer, negative (treated as 0), or floating-point and greater than 2147 (converted to the nearest integer), the `sleep()` system call is used. Otherwise, *s* is converted to the nearest microsecond (multiplied by one million and rounded to an integer) and the `usleep()` system call is used. The result of the function is the result of the system call.

system *sys.system{v}*

Argument and Result

The argument is a character vector. The result is a scalar integer.

Definition

This function is equivalent to the `system()` system call, for executing character strings as if they were commands typed at a terminal. The result of the function is the result of the system call. This result depends upon the particular system - *`sys.system`* is ***highly nonportable***. If you need advice about the use of this function, you should usually ask someone who knows a lot about Unix, and not necessarily someone who knows a lot about A+.

Note that `sh` is invoked by default, not `ksh`, `csh`, or whatever; in particular, this means that `~` in path names is not treated as a "magic" character. If you need this or some other property of one of these shells, invoke that shell explicitly.

access *`sys.access{f;m}`*

Arguments and Result

The left argument *`f`* is a character vector and the right argument *`m`* is a symbol scalar or vector, or integer scalar. The result is an integer scalar.

Definition

This function is equivalent to the `access()` system call, for determining the accessibility of a file. The left argument *`f`* holds a path name of a file (see "Path Names"), and the right argument *`m`* holds manifest constants (see "Manifest Constants"). The result of the function is the result of the system call.

File Stats *`sys.astat{c}`*

Argument and Result

The argument *`c`* is a character array. The result is an integer array of shape `(¯1↓⍴c),13`.

Definition

The value of this function contains a variety of information produced by the `stat()` system call. If the argument *`c`* is a scalar or vector, it contains a path name of a file, and the result contains 13 entries for that file. If the argument has rank greater than 2, each cell of rank 1, i.e., each vector along the last axis, is a path name of a file, and the corresponding cell of rank 1 in the result contains 13 entries for that file. Trailing blanks are removed from the path names in both cases (see "Path Names"). The meanings of these entries in terms of the structure of the stat result are given in the table "sys.astat and sys.alstat Results in Terms of stat System Call Result". If the `stat()` system call fails for a path name, the corresponding entries in the result are all zero.

sys.astat and sys.alstat Results in Terms of stat System Call Result

A+ Index	stat Entry	A+ Index	stat Entry	A+ Index	stat Entry

0	st_dev	5	st_gid	9	st_mtime
1	st_ino	6	st_rdev	10	st_ctime
2	st_mode	7	st_size	11	st_blksize
3	st_nlink	8	st_atime	12	st_blocks
4	st_uid				

close `sys.close{f}`

Argument and Result

The argument `f` and the result are scalar integers.

Definition

This function is equivalent to the `close()` system call, for deleting the file descriptor `f`. The result of the function is the result of the system call.

create `sys.creat{f;m}`

Arguments and Result

The left argument `f` is a character vector, and both the right argument `m` and result are scalar integers.

Definition

This function is equivalent to the `creat()` system call, for creating a new file or rewriting an existing one. The left argument `f` is the path name of a file (see "Path Names"), and the right argument `m` is the permissions mode (see "File Permissions Mode"). The result of the function is the result of the system call.

File Size `sys.filesize{f}`

Argument and Result

The argument `f` is a character vector. The result is an integer scalar.

Definition

The result is the size in bytes of the open file whose path name is in the argument `f` (see "Path Names"). This function uses the `fstat()` system call, which provides a variety of information about (possibly) open files.

flock `sys.flock{f;o}`

Arguments and Result

The left argument *f* is a file descriptor, and the right argument *o* is a symbol scalar or vector, or integer scalar. The result is an integer scalar.

Definition

This function is equivalent to the `flock()` system call, for applying or removing advisory locks on a file. The left argument *f* holds the descriptor of a file. The right argument *o* holds manifest constants (see "Manifest Constants") for the second argument of `flock`. The result of the function is the result of the system call. For some releases and some architectures, the file must be opened for writing in order for this function to succeed.

fsync sys.fsync{f}

Argument and Result

Both the argument and result are integer scalars.

Definition

This function is equivalent to the `fsync()` system call, for synchronizing the incore copy of a file with the copy on disk. The argument *f* is a file descriptor. The result of the function is the result of the system call.

getdtablesize *sys.getdtablesize{}*

Arguments and Result

The result is an integer scalar.

Definition

This function is equivalent to the `getdtablesize()` system call, which returns the maximum number of file descriptors. The result of the function is the result of the system call.

lseek *sys.lseek{f;o;w}*

Arguments and Result

All arguments and the result are integer scalars.

Definition

This function is equivalent to the `lseek()` system call, for setting the pointer in the open file pointed to by the file descriptor *f*. The arguments of the function are in the same order as they will be in the call to `lseek()`. The result of the function is the result of the system call.

For the argument *w*, use 0 for `SEEK_SET`, 1 for `SEEK_CUR`, and 2 for `SEEK_END`.

open *sys.open{f;l;m}*

Arguments and Result

The argument *f* is a character vector or symbol. The argument *l* is a symbol vector or scalar, or an integer scalar. The argument *m* and the result are integer scalars.

Definition

This function is equivalent to the `open()` system call, which opens a file for reading, or writing, or both. The argument *f* holds the path name of a file (see "Path Names"), the argument *l* holds the manifest constants flags for the call to `open()` (see "Manifest Constants"), and the argument *m* is the permissions mode (see "File Permissions Mode"), perhaps modified by the process's `umask` value - see the `open()` man page. The result of the function is the result of the system call.

pathfind *sys.pathfind{v;p;f;m}*

Arguments and Result

The arguments *v*, *p*, and *f* are character vectors. The argument *m* is an integer scalar. The result is either a character vector or the Null.

Definition

This function searches for the file named in the argument *f*, and returns its full path name if it is found and if it can be opened with the permissions specified by the argument *m* (see "File Permissions Mode"). Otherwise, it returns the Null. If the argument *v* holds an environment variable name, then the value of that variable specifies the search path. Otherwise, the argument *p* specifies the search path.

File to Character Vector *sys.read{f;a;n}*

Arguments and Result

The arguments *f* and *n*, and the result, are integer scalars. The argument *a* is a character vector.

Definition

This function is nearly equivalent to the `read()` system call, for reading from the object specified by a file descriptor. The argument *f* is a file descriptor, *n* is the number of bytes to be read, and *a* is an A+ character vector of length at least *n*. The effect is to read at most *n* bytes from the file pointed to by *f* and place them in the value of *a*. The result is the actual number of bytes read.

Warning! In most uses of *sys.read*, the argument *a* is a variable, and this variable exhibits the same behavior under ordinary assignment as mapped files. Consider the sequence:

```
x←100⍴' '
y←x
sys.read{fd;x;#x}
```

At this point the value of *y* also has the same 100 characters that were read from the file into *x*; that is, *y*≡*x* both before and after the *sys.read* evaluation.

rename *sys.rename{f;g}*

Arguments and Result

The arguments *f* and *g* are character vectors. The result is an integer scalar.

Definition

This function is equivalent to the `rename()` system call, for renaming links. The arguments of the function are in the same order as they will be in the call to `rename()`. The result of the function is the result of the system call.

truncate *sys.truncate{f;n}*

Arguments and Result

The left argument *f* is a character vector. The right argument *n* and the result are scalar integers.

Definition

This function is equivalent to the `truncate()` system call, for specifying the size of a file in bytes. The left argument *f* holds the path name of a file and the right argument *n* is the file size, in bytes. The result of the function is the result of the system call.

ftruncate *sys.ftruncate{f;n}*

Arguments and Result

Both arguments and the result are integer scalars.

Definition

This function is equivalent to the `ftruncate()` system call, for specifying the size of a file in bytes. The left argument *f* is a file descriptor and the right argument *n* is the file size, in bytes. The result of the function is the result of the system call.

umask *sys.umask{m}*

Argument and Result

Both the argument and result are integer scalars.

Definition

This function is equivalent to the `umask()` system call, for changing the session default file creation mask. The result of the function is the current setting, so later restoration is possible - including an immediate restoration when the aim is only to check the value. The new setting is the exclusive OR of *m* and 666 (octal) for a file or 777 for a directory. Thus an argument of 002 gives complete access to the group and read (and directory search) access to others.

Execute

```
sys.umask ↓sys.umask 0;
```

to display the current setting while leaving it unchanged.

File Update Time `sys.updtime{f}`

Argument and Result

The argument is a character vector. The result is a one-element integer vector.

Definition

The argument f holds the path name of a file (see "Path Names"). The result is the time of the latest update of the file, measured in seconds since 00:00:00 GMT, January 1, 1970. This function uses the `fstat()` system call, which provides a variety of information about (possibly) open files.

write `sys.write{f;a;n}`

Arguments and Result

The arguments `f` and `n`, and the result, are integer scalars. The argument `a` is a character vector.

Definition

This function is nearly equivalent to the `write()` system call, for writing in a file. The argument `f` is a file descriptor , `n` is the number of bytes to be written, and `a` is an A+ character vector of length at least `n`. The effect is to write at most `n` bytes from `a` in the file pointed to by `f`. The result is the actual number of bytes written.

For example, `sys.write{1;'hey';#'hey'};` will write 'hey' to stdout, and `sys.write{2;'ho';#'ho'};` will write 'ho' to stderr, both without a newline. The file descriptors 1 and 2 are already opened when you start A+; there is no need to call `sys.open`.

closelog `sys.closelog{}`

Result

The result is the Null.

Definition

This function is equivalent to the `closelog()` system call, for closing the system log.

openlog `sys.openlog{c;l;f}`

Arguments and Result

The argument `c` is a character vector. The arguments `l` and `f` are symbol vectors or scalars, or integer scalars. The result is the Null.

Definition

This function is equivalent to the `openlog()` system call, for initializing the system log file. The arguments of the function are in the same order as they will be in the call to `openlog()`. The arguments `l` and `f` hold manifest constants (see "Manifest Constants").

syslog *sys.syslog{a;m}*

Arguments and Result

The left argument *a* is a symbol scalar or vector, or an integer scalar. The right argument *m* is a character vector.

Definition

This function is equivalent to the `syslog()` system call, for logging messages. The arguments of the function are in the same order as they will be in the call to `syslog()`. The argument *m* holds manifest constants (see "Manifest Constants").

errno *sys.errno{}*

Result

The result is a scalar integer.

Definition

The result is identical to the global `errno` in the Unix system, which is set by the last system call. About obtaining error descriptions, see *sys.errsym*, below.

Error Symbol *sys.errsym{n}*

Argument and Result

The argument *n* is a scalar integer. The result is a scalar symbol.

Definition

The argument is an error number, as produced by *sys.errno* (above). The result is text describing the error.

perror *sys.perror{v}*

Argument and Result

The argument *v* is a character vector. The result is the Null.

Definition

This function is equivalent to the `perror()` system call, which writes a short error message to standard error describing the error whose number is in `errno` (see *sys.errno*, above).

Readlink *sys.areadlink{f}*

Argument and Result

The argument is a character vector. The result is either a character vector or the scalar integer -1.

Definition

The argument holds a path name of a file (see "Path Names"). If the file is a symbolic link, the result is a character vector holding the name of the file it references. If the file named in the argument is not a symbolic link, the result is -1.

Linked File Stats `sys.alstat{c}`

Argument and Result

The argument `c` is a character array. The result is an integer array of shape `(¯1↓⍴c),13`.

Definition

This function is the same as `sys.astat`, except that it uses the `lstat()` system call instead of `stat()`.

link `sys.link{a;b}`

Arguments and Result

The arguments `a` and `b` are character vectors. The result is an integer scalar.

Definition

This function is equivalent to the `link()` system call, for establishing file links. The arguments, which are path names (see "Path Names") of the function are in the same order as they will be in the call to `link()`.

The result of the function is the result of the system call.

symlink `sys.symlink{a;b}`

Arguments and Result

The arguments `a` and `b` are character vectors. The result is an integer scalar.

Definition

This function is equivalent to the `symlink()` system call, for creating symbolic links. The arguments of the function are in the same order as they will be in the call to `symlink()`. The result of the function is the result of the system call.

unlink `sys.unlink{f}`

Argument and Result

The argument `f` is a character vector. The result is an integer scalar.

Definition

This function is equivalent to the `unlink()` system call, for removing the directory entry named by the path name `f` (see "Path Names") and decrementing the link count on the file named by `f`. The result of the function is the result of the system call.

Domain Name *sys.getdomainname{}*

Result

The result is a character vector.

Definition

The result is the name of the domain of the current network domain.

getgid *sys.getgid{}*

Result

The result is an integer scalar.

Definition

This function is equivalent to the `getgid()` system call, which returns the (real) group ID of the current process. The result of the function is the result of the system call.

Host Name *sys.gethostname{}*

Result

The result is a character vector.

Definition

The result is the name of the current host machine.

getpid *sys.getpid{}*

Result

The result is an integer scalar.

Definition

This function is equivalent to the `getpid()` system call, which returns the process ID of the current process. The result of the function is the result of the system call.

getppid *sys.getppid{}*

Result

The result is an integer scalar.

Definition

This function is equivalent to the `getppid()` system call, which returns the process ID of the parent of the current process. The result of the function is the result of the system call.

geteuid *sys.geteuid{}*

Result

The result is an integer scalar.

Definition

This function returns the effective user id, corresponding to the name returned by *$whoami*. See getuid, User Name, and User Name from ID, below.

getuid *sys.getuid{}*

Result

The result is an integer scalar.

Definition

This function is equivalent to the *getuid()* system call, which returns the real user id of the current process. The result of the function is the result of the system call.

User Name *sys.getusername{}*

Result

The result is a character vector, or the null.

Definition

The result is the name of the logged-in user: the "real" user name, in contrast to the "effective" user name, which is returned by *$whoami*. These two names can differ when *su* or a *setuid script* is used. *sys.getusername{}* calls the c library function *pwd=getpwuid(getuid())* and returns the *pwd->pw_name* member of *pwd*. Null is returned if the requested entry is not found, or on an error or EOF; if null is returned, you can retry in a few seconds and see whether the problem (perhaps an updating) is gone.

User Name from ID *sys.username{i}*

Result

The result is a character vector.

Definition

This function returns the user name corresponding to the user id *i*. See geteuid, getuid, and User Name, above.

Directory Entries *sys.agetdents{f}*

Argument and Result

The argument *f* is a character vector. The result is a character matrix.

Definition

The argument *f* holds a path name to a directory (see "Path Names"). The rows of the result are the names of the files in that directory. The special files named `.` and `..` are removed from the list.

chdir *sys.chdir{s}*

Argument and Result

The argument *s* is a character vector. The result is an integer scalar.

Definition

This function is equivalent to the `chdir()` system call, for changing the process's working directory. The argument *s* holds the path name of a directory (see "Path Names"). The result of the function is the result of the system call. The PWD environment variable is set.

The A+ system function *_cd* should be used in place of this function.

mkdir *sys.mkdir{f;m}*

Arguments and Result

The left argument *f* is a character vector, and the right argument *m* is an integer scalar.

Definition

This function is equivalent to the `mkdir()` system call, for creating directories. The left argument *f* holds the path name of a file (see "Path Names"), and the right argument *m* is the permissions mode (see "File Permissions Mode"). The result of the function is the result of the system call.

rmdir *sys.rmdir{f}*

Argument and Result

The argument *f* is a character vector. The result is an integer scalar.

Definition

This function is equivalent to the `rmdir()` system call, for removing directories. The argument *f* holds a path name of a file (see "Path Names"). The result of the function is the result of the system call.

chmod *sys.chmod{s;m}*

Arguments and Result

The left argument *s* is a character vector. The right argument *m* and the result are integer scalars.

Definition

This function is equivalent to the `chmod()` system call, for changing permissions mode of a file. The left argument *s* holds the path name of a file (see "Path Names"), and the right

argument *m* holds the permissions mode (see "File Permissions Mode"). The result of the function is the result of the system call.

fchmod *sys.fchmod{f;m}*

Arguments and Result

The left argument *f*, the right argument *m*, and the result are all integer scalars.

Definition

This function is equivalent to the `fchmod()` system call, for changing permissions mode of a file. The left argument *f* is a file descriptor, and the right argument *m* holds the file permissions mode (see "File Permissions Mode"). The result of the function is the result of the system call.

chown *sys.chown{s;m;n}*

Arguments and Result

The argument *s* is a character vector. The arguments *m* and *n*, and the result, are integer scalars.

Definition

This function is equivalent to the `chown()` system call, for changing owner and group of a file. The argument *s* holds the path name of a file (see "Path Names"), and the arguments *m* and *n* are the specifications for the owner and group, respectively. The result of the function is the result of the system call.

Only a superuser can change the owner of a file. An owner can change the group of a file to one of which he is a member.

fchown *sys.fchown{f;m;n}*

Arguments and Result

The arguments *f*, *m*, and *n*, and the result, are all integer scalars.

Definition

This function is equivalent to the `fchown()` system call, for changing the owner and group of a file. The argument *f* is a file descriptor, and the arguments *m* and *n* are the specifications for the owner and group, respectively. The result of the function is the result of the system call.

Only a superuser can change the owner of a file. An owner can change the group of a file to one of which he is a member.

CPU Time *sys.cpu{}*

Result

The result is a four-element integer vector.

Definition

This function uses the `times()` C-library routine to produce a four-element vector whose elements are (in order): user time of the current process; system time of the current process; user time for the children of the current process; system time for the children.

Current GMT *sys.tsgmt*{}

Result

The result is a seven-element integer vector.

Definition

The result is the current Greenwich mean time, GMT, represented as: year, month, day, hour, minute, second, millisecond.

Current Local Time *sys.ts*{}

Result

The result is a seven-element integer vector.

Definition

The result is the current local time, represented as: year, month, day, hour, minute, second, millisecond.

Time of Day *sys.gettod*{*x*}

Argument and Result

The argument *x* is an integer vector. The result is a two-element integer vector.

Definition

This function uses the `gettimeofday()` system call. If the argument *x* has two elements then the first element represents minutes west of Greenwich and the second element is the Daylight Savings flag. If *x* does not have two elements, `gettimeofday()` is called with the NULL pointer as its second argument. The result of the function is the result of the system call.

GMT *sys.ts*1*gmt*{*c*}

Argument and Result

The argument is an integer scalar. The result is a seven-element integer vector.

Definition

The argument represents some number of seconds since 00:00:00 GMT, January 1, 1970. The result is the equivalent GMT, represented as: year, month, day, hour, minute, second, 0.

Local Time *sys.ts*1{*c*}

Argument and Result

The argument is an integer scalar. The result is a seven-element integer vector.

Definition

The argument represents some number of seconds since 00:00:00 GMT, January 1, 1970. The result is the equivalent local time, represented as: year, month, day, hour, minute, second, 0.

Time in Seconds `sys.mkts1{x}`

Argument and Result

The argument is an integer vector of length 7. The result is a scalar integer.

Definition

The argument should be a local time in the form years, months, days, hours, minutes, seconds, microseconds. The argument is changed to `7↑¯1↓x` - i.e., the given time with the microseconds ignored.

The result is this time converted to the number of seconds since 00:00:00 GMT, Jan. 1, 1970. If the conversion cannot be made (1930 ... or 2094 ... , for example), the result is -1.

For an argument *x* that is in the appropriate range, `sys.mkts1 sys.ts1 x` is *x*.

Seconds in Epoch `sys.secs_in_epoch{}`

Result

The result is an integer scalar.

Definition

The result is the current time measured in seconds since 00:00:00 GMT, January 1, 1970.

Note that the result, which is identical to `0#sys.gettod{0}`, can be used as an argument to `sys.ts1` or `sys.ts1gmt`.

Reset Time Zone `sys.tzset{}`

Definition

Resets the time zone to the current local time zone. The result is Null.

dup `sys.dup{f}`

Argument and Result

The argument *f* and the result are integer scalars.

Definition

This function is equivalent to the `dup()` system call, for duplicating the file descriptor *f*. The result of the function is the result of the system call.

dup2 `sys.dup2{f;g}`

Arguments and Result

Both the left argument *f* and the right argument *g* are integer scalars.

Definition

This function is equivalent to the `dup2()` system call, for duplicating the file descriptor *f* and specifying its value to be that of the file descriptor *g*. The result of the function is the result of the system call.

Read from a Socket *sys.aread{f;w}*

Arguments and Result

The left argument *f* and the right argument *w* are integer scalars. The result is a simple character, integer, or floating-point array.

Definition

The left argument *f* is the file descriptor of an internet stream socket. The effect of this function is to read the simple character, integer, or floating-point array from the connected process. If the right argument *w* is 0 then the system call `select()` is used with the zero timer to determine whether the file descriptor is ready to read. If not, execution ends and the result of this function is the Null. Otherwise, or if the value of *w* is not 0, the `read()` system call is invoked as often as necessary to build the array. If the function succeeds, the result is that array. If it fails for any reason, the result is the Null.

Read from a Socket and Return the Status *sys.areadstat{f;w;s}*

Arguments and Result

The arguments *f*, *w* and *s* are integer scalars. The argument *s* may be modified directly by this function (see "Functions that Modify Arguments"). The result is a simple character, integer, or floating-point array.

Definition

This function is the same as *sys.aread{f;w}*, except that if the read fails the value of the argument *s* is set to -2 for an EWOULDBLOCK error, and -1 for all other errors.

Read from a Socket within a Time Interval *sys.areadwait{f;s;u}*

Arguments and Result

The arguments *f*, *s*, and *u* are integer scalars. The result is a simple character, integer, or floating-point array.

Definition

This function is the same as *sys.aread{f;0}*, except that it uses the `select()` system call with the timer set to *s* seconds and *u* microseconds, instead of 0.

getsockopt *sys.getsockopt{s,l,on,ov,ol}*

Arguments and Result

The arguments *s*, *l*, and *on*, and the result, are integer scalars. The arguments *ov* and *ol* are integer vectors. Both these arguments can be modified directly by the function (see "Functions that Modify Arguments").

Definition

This function is equivalent to the `getsockopt()` system call, for manipulating options associated with a socket. The arguments of the function are in the same order as they will be in the call to `getsockopt()`. The result of the function is the result of the system call.

setsockopt *sys.setsockopt{s,l,on,ov,ol}*

Arguments and Result

The arguments *s*, *l*, *on*, and *ol*, and the result, are integer scalars. The argument *ov* is an integer vector that can be modified directly by the function (see "Functions that Modify Arguments").

Definition

This function is equivalent to the `setsockopt()` system call, which manipulates options associated with a socket. The arguments of the function are in the same order as they will be in the call to `setsockopt()`. The result of the function is the result of the system call.

Socket Accept *sys.sockaccept{f;w}*

Arguments and Result

Both arguments *f* and *w*, and the result are scalar integers.

Definition

The argument *f* is the file descriptor of an internet stream socket. This is a server function whose effect is to accept connections from clients. The argument *w* is a wait flag. If *w* is 1 then the function blocks, i.e., does not return, until a connection is made. Otherwise, it returns immediately. If a connection is made, the result is the file descriptor of the connected socket. The result is -1 in case of an error, or -2 if *w* is 0 and no connection is made. This function uses the following system calls: `accept()`, `select()`, and `setsockopt()` (setting `SO_KEEPALIVE`).

Socket Block *sys.sockblock{f;b}*

Arguments and Result

Both arguments *f* and *b* are scalar integers. The result is the Null.

Definition

The argument *f* is a file descriptor of an internet stream socket. The effect of this function is to set blocking on the socket if the blocking flag *b* is 1, and to set no blocking if *b* is 0. If blocking is set to 1 then *sys.awrite* does not return until the A+ array is entirely written, even though

this may require the system to break the array into pieces and send them separately. This function uses the system call `ioctl()`.

Socket Connect *sys.sockconnect{h;p}*

Arguments and Result

The left argument *h* is a character vector. The right argument *p* is an integer scalar. The result is an integer scalar.

Definition

The left argument *h* holds the name of a host machine, and the right argument *p* is an internet TCP port. This is a client function whose effect is to connect to an internet stream socket on that port and host; a listening server is expected to be present. The result is the file descriptor of the connected socket if the connection is successful, and -1 otherwise. This function uses the following system calls: `connect()`, `setsockopt()` (setting `SO_KEEPALIVE`), and `socket()`.

Socket Listen *sys.socklisten{p}*

Argument and Result

The argument *p* and the result are scalar integers.

Definition

This is a server function whose effect is to create an internet stream socket for listening, and bind it to the internet TCP port specified by the argument *p*. The result is the file descriptor of the listening socket, or -1 if an error occurs. This function uses the following system calls: `bind()`, `listen()`, `setsockopt()` (setting `SO_REUSEADDR`), and `socket()`.

Socket ForkExec *sys.sfe{v;a}*

Arguments and Result

The left argument *v* is a character vector and the right argument *a* is a simple integer vector.

Definition

This function opens a bidirectional pipe, forks a new process, and makes the system call `execvp()`.

The arguments *v* and *a* hold valid first and second parameters to `execvp()`, respectively. In particular, the right argument is a vector of pointers to character strings, and its last element is 0 (if *x* is an A+ character vector, then *c.ptr x* is the pointer to the character string value of *x*). The result is the file descriptor for stdin and stdout of the forked process.

Delete Defunct Children *sys.zombiekiller{}*

Result

This function returns a count of how many processes it reaped, for lack of anything more useful.

Definition

This function deletes any defunct child processes of the current A+ session.

Write to a Socket `sys.awrite{f;a}`

Arguments and Result

The left argument `f` is a scalar integer, and the right argument `a` is any simple character, integer, or floating-point array. The result is a scalar integer.

Definition

The left argument `f` is the file descriptor of an internet stream socket. The effect of this function is to send the A+ array `a` to the connected process. The result is 0 if the function is successful. The result is -2 if an EWOULDBLOCK error occurs; all other errors produce a result of -1. This function uses the `write()` system call.

Note that if a blocking error occurs, then an initial segment of the array `a` was sent.

Select `sys.aselect{r;w;x;t}`

Arguments and Result

The arguments are all integer scalars or vectors, with `t` of length 0, 1, or 2. The result is a five-element nested vector.

Definition

This function calls the `select()` system call with the arguments `r`, `w`, and `x` as the read, write, and exceptional condition file descriptors, respectively, to determine which are ready. If the argument `t` is empty then select is called with the zero timer. Otherwise, the first element of `t` is seconds and the second element, if present, is microseconds. The width parameter of the `select()` call is computed to be one plus the largest value among the file descriptors in `r`, `w`, and `x`.

The first element of the result is the return code from `select()`. The second element is the value of errno if that return code is negative, and 0 otherwise. The last three elements are the modified `r`, `w`, and `x` lists produced by `select()`.

fcntl `sys.fcntl{f;c;a}`

Arguments and Result

The arguments `f` and `a`, and the result, are integer scalars. The argument `c` is a symbol scalar or integer scalar. The argument `a` may be modified directly by this function (see "Manifest Constants").

Definition

This function is equivalent to the `fcntl()` system call, for performing a variety of functions on file descriptors. The argument *f* is a file descriptor, and the argument *c* is a manifest constant (see "Functions that Modify Arguments") designating the function for the system call. The argument *a* contains additional information, or is modified with new information, depending on *c*. The result of the function is the result of the system call.

ioctl *sys.ioctl{f;c;a}*

Arguments and Result

The arguments *f* and *a*, and the result, are integer scalars. The argument *c* is a symbol scalar or integer scalar. The argument *a* may be modified directly by this function (see "Functions that Modify Arguments").

Definition

This function is equivalent to the `ioctl()` system call, for performing special functions on an object with the open file descriptor *f*. The argument *c* is a manifest constant (see "Manifest Constants") designating the function for the system call. The argument *a* contains additional information, or is modified with new information, depending on *c*. The result of the function is the result of the system call.

read *sys.readinto{f;b;n}*

Arguments and Result

The arguments *f*, *b* , and *n*, and the result, are integer scalars.

Definition

This function is equivalent to the `read()` system call, for reading from a file. The argument *f* is a file descriptor, *n* is the number of bytes to be read, and *b* is a pointer to a character string of length at least *n*. The effect is to read at most *n* bytes from the file pointed to by *f* and place them in the character string pointed to by *b*. The result is the actual number of bytes read.

11. The t Context

Introduction

In order to establish the t context, it is necessary to perform a `$load t`.

Just as the screen management system, the s context, provides the support for displaying tables on screens, the t context provides support for creating and manipulating tables in ways conforming to database queries. Like tables in the table display class (s-tables), t-tables are composed of *column*, or *field*, variables. However, t-tables are more complex. Each t-table is represented by an A+ context with the same name as the table, containing the column variables of the table as well as a set of variables and dependencies describing various characteristics of

the table and its relationships to other tables; this set is described in the table "t-Created Variables, Dependencies Common to All Table Contexts". For example, the variable `_T` is a list of the names of the column variables as symbols, and appears in every t-table context. Programmers can define their own variables and dependencies in terms of this set, thereby integrating t-tables into their applications and synchronizing changes in data. The t-generated objects in the table contexts are called *table variables* and *table dependencies*. They, and column variables, are referred to here by their unqualified names unless it is necessary to do otherwise.

With regard to database maintenance and manipulation, t is the manipulation component, i.e., the computational engine that implements query languages. It can be helpful to think of the t functions in terms of relational operations, which are conceptually simple and have proven to be an effective, complete set for implementing database queries. Even though there is not an exact fit between t functions and relational operations, they are comparable. See "The Relational Set Operations: Union, Intersection, and Difference" for examples of defining relational operations in terms of t.

t-Tables: Base Tables and Views

Every t-table is an A+ context; the name of the table is the name of the context. As the last sentence illustrates, t-tables will be referred to simply as tables where the context permits. The function `t.open` is the means of creating and initializing t-tables from mapped files, text files, and ordinary variables and dependencies. Tables created by `t.open` are called *base tables*. Other tables, called *views* or *derived tables*, are created from base tables and other views. In the process of creating a view, the view is called the *target*, and the table(s) from which it is created the *source(s)*. A view of the table `X`, for example, can be created as the result of applying a selection function such as `t.only` to `X`, or the summarization function `t.group` (for which `t.break` is a synonym). At the time of creation the view has no columns; it is populated by sending it some or all of the columns of `X` with the function `t.send`. The selection functions can also operate on views in place, i.e., without creating a second view to hold the selection; they cannot be applied to base tables in place. The function `t.define` is used to append new fields to existing tables. The function `t.link` is the means for relating one table to another; it includes a variation of `t.group`, and is the most distinctive feature of t.

Not all t-tables are valid s-tables. The definition of `t.open` gives the full range of possibilities for t-tables. See `t.table` for extracting the subset of a t-table that can be displayed on the screen.

Table and Column Names

Names of tables and table columns always appear as symbols in A+ expressions, but elsewhere in this chapter they often appear in ordinary font.

Tables are contexts, and while it is common to refer to objects in contexts by their fully qualified names, this is not the case for table columns. Unqualified column names should always be used in the arguments of the t functions except for columns outside the table context; the table will be determined from context.

Column names cannot begin with an underbar (_).

Table Row and Column Domains

A table column is an A+ array whose *length* is its item count. If all columns of a table have the same length, one can speak of the *length of the table* as the common length of its columns. Like s-tables, t-tables need not have columns whose lengths are all the same, although that is most

generally useful. It is assumed in all examples in this chapter that all columns of a table have the same length. When that assumption is true, the length of the table is maintained in the table variable `_N`. Otherwise, `_N` is the number of items in the column named by `0#_T`, where `_T` is the table variable holding a list of all column names.

If c1, c2, ... , cn are columns in a table then the list `(i#`c1;i#`c2;...;i#`cn)` is called their `i`th *row*, and if these are all the columns, then the list is called the `i`th *row of the table*.

The *domain* of a table column consists of the shape of its items and its general type; two table columns are in the same domain if they are a valid left argument, right argument pair for the Catenate primitive. More generally, two sets of table columns `c1, c2, ... , cn` and `d1, d2, ... , dn` are said to be in the *same domain* if every pair `ci` and `di` are in the same domain.

See "Row and Column Selection", below, for more on the relationship between base tables and views.

It is generally assumed throughout this chapter, unless something is said to the contrary, that tables have distinct rows. This assumption is not restrictive in practice: when tables are constructed, intermediate results may have duplicate rows, but otherwise it is very rare for duplicate rows to be of use.

About the Examples

The tables and A+ expressions for creating the examples can be found online. The mapped files for these tables are created by the A+ script `/usr/local/aplus-fsf-4.18/doc/tutorials/t.tutorial/files.+` and the t tables are loaded by the A+ script `model.+` in the same directory. To create your own copies of the mapped files and experiment with variations of the t tables, make copies of these scripts and change their directory variables. There is also a script that recreates the examples in the text. See the `README` file in that directory for more information.

Only the tables named `employees` and `departments` are actually used in this document (see the base tables figure). They are created using the t function `t.open`, as follows:

```
$load t
directory←"/usr/local/aplus-fsf-4.18/doc/tutorials/t.tutorial/files/"
(`departments;directory) t.open (
                          `dept_no;"0Idept.dept_no";
                          `dept_name;"¯1Idept.dept_name";
                          `mgr;"0Idept.mgr";
                          `supdept;"0Idept.supdept")
(`employees;directory) t.open (
                          `emp_no;"0Iemp.emp_no";
                          `emp_name;"¯1Iemp.emp_name";
                          `dept_no;"0Iemp.dept_no";
                          `hire_date;"0Iemp.hire_date";
                          `sex;"0Iemp.sex";
                          `salary;"0Iemp.salary";
                          `commission;"0Iemp.commission")
employees.emp_name←⊢⊃⍋¨employees.emp_name
```

The Base Tables:

employees

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
10	Haas	A00	650101	f	52750	
20	Thompson	B01	731010	m	41250	
30	Kwan	C01	750405	f	40175	
50	Geyer	E01	490817	m	38250	
60	Stern	D11	730914	m	36250	
70	Pulaski	D21	800930	f	36170	
90	Henderson	E11	700815	f	35750	
110	Lucchessi	A00	580516	m	38170	
130	Quintana	C01	710728	f	33800	
140	Nicholls	C01	761215	f	35420	
150	Adamson	D11	720212	m	30280	
170	Yoshimura	D11	780915	m	28680	
190	Walker	D11	740726	m	20450	
230	Jefferson	D21	661121	m	22180	
240	Marino	D21	791205	m	33760	
290	Parker	D31	800530	m	15340	4780
300	Smith	D31	720619	m		17750
310	Setright	D31	640912	f	15900	3200

departments

dept_no	dept_name	mgr	supdept
A00	Spiffy Computer Service Div.	10	
B01	Planning	20	A00
C01	Information Center	30	A00
D01	Development Center		A00
E01	Support Services	50	A00
D11	Manufacturing Systems	60	D01
D21	Administration Systems	70	D01
D31	Order Processing Systems		D01
E11	Operations	90	E01
E21	Software Support		E01

Row and Column Selection

Row and column selection in t is done using `t.only` and `t.send`. For example, the `employees` table in the figure can be restricted to departments D11, D21, and D31 as follows:

```
`employees `r_view t.only 'dept_no∈3 3⍴"D11D21D31"'
`employees `r_view t.send ⍬
```

Selection of the rows is done by `t.only` and is similar to the A+ primitive function called Replicate. The left argument specifies the source table on which the selection is made and the target, in which the selection is realized. The right argument is a character vector holding a boolean-valued expression that specifies the rows to be selected. The view can already exist or not; if it already exists then its current definition is closed (see `t.close`) and replaced with the new one.

Once `t.only` is executed, the table `r_view` has been created but has no columns. To complete the row selection, send the columns of the source to the target. The right argument `⍬` to

`t.send` indicates that all columns are to be sent. See the selection figure for the result of this selection.

As for column selection, columns `emp_no`, `emp_name`, and `dept_no` of the employees table can be selected as follows:

```
`employees `p_view t.only ()
`employees `p_view t.send `emp_no `emp_name `salary
```

The right argument `()` to `t.only` indicates that all rows are selected, and the right argument to `t.send` specifies the columns to send to the selection table, p_view. See the figure.

Row and column selection can be combined. For example, a view of the `emp_no`, `emp_name`, and `salary` columns for all employees in departments D11, D21, and D31 is:

```
`employees `rp_view t.only 'dept_no∈3 3⍴"D11D21D31"'
`employees `rp_view t.send `emp_no `emp_name `salary
```

There is a utility function in t named `t.in` that allows a more Englishlike specification of a set of values in a selection, in this case`"D11,D21,D31"` in place of `3 3⍴"D11D21D31"`. See "Selectors" for more on this topic.

There are several other points to be made about row selection. First of all, it is not necessary to do the row selection all at once. For example, after making the department selections above, department E01 can be added to the selection as follows:

```
`employees `rp_view t.also 'dept_no∈1 3⍴"E01"'
```

When both the source and target are specified in the left argument to `t.also`, the selection is evaluated on the source table. However, if only the target is specified in that left argument, the selection is evaluated on the complement of the target (those rows that are not currently selected), and the result is appended to the target. In the latter case, any columns named in the selection must have been sent to the target before the selection is made.

For example, the selection using `t.also` above can be rephrased as:

```
`employees `rp_view t.send `dept_no
`rp_view t.also 'dept_no∈1 3⍴"E01"'
```

Note that this formulation tends to be more efficient than the one above, which refers to all rows of the source table.

Applications of `t.also` have the effect of ORing the selection specified in its right argument with any previous selections made with this function. If a subsequent selection is done with `t.only`, it will replace the existing one.

By default, selections ultimately refer to the entire source table of a view, or to the entire complement of a view in its source table. For example, if the following selection is now executed:

```
`employees `rp_view t.send `emp_no
`rp_view t.also 'emp_no∈60 90'
```

then one more row will appear in *rp_view*, the one for the employee with employee number 90, because that employee's department is not one of D11, D21, D31, and E01. If, however, the view had been fixed before the employee number selection was made, as in:

```
`rp_view t.fix 1
```

then the employee number selection would have applied only to the rows in the view table at the time that *t.fix* is executed, and consequently, no new rows would appear. The logical effect of executing *t.fix* with a right argument of 1 is to And the selections made before its execution with those that follow, and a practical effect is to make subsequent selections more efficient by limiting their scope.

How t Manages Row and Column Selection

Every view table created by *t.only* has a table variable *_V*, which is a vector of row indices in the source table of the view that reflects any row selections done on the source; the *i*th item of any column sent to the view will be the *_V[i]*th item of the source column. The value of *_V* is kept current with all selections and reordering of the rows in the view.

When a column *col* is sent from a source table *src* to a target table *tar*, the target column is, in the default case, defined by the following dependency in the *tar* context:

```
col:_V#src.col
```

Actually, the dependency is somewhat more complicated, and more like:

```
col:_index{_V;src.col;NA}
```

where *NA* denotes the appropriate NA value to be used when indices in *_V* are out of range; see the definition of *t.index*, and the *`na* attribute in the definition of *t.open*. This default dependency definition can be modified when the source column is sent; see "Dependency Frames" and the definition of *t.send*.

The dependencies provide mappings from source columns to view columns. There are also mapping provided by t in the other direction, from view columns to source columns. These are accomplished by putting callbacks on the view columns which make the appropriate updates to source columns whenever they are called.

Column selections are similarly maintained in the table variable *_T*.

Row and Column Selection:

r_view

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
60	Stern	D11	730914	m	36250	
70	Pulaski	D21	800930	f	36170	
150	Adamson	D11	720212	m	30280	
170	Yoshimura	D11	780915	m	28680	
190	Walker	D11	740726	m	20450	
230	Jefferson	D21	661121	m	22180	
240	Marino	D21	791205	m	33760	
290	Parker	D31	800530	m	15340	4780
300	Smith	D31	720619	m		17750
310	Setright	D31	640912	f	15900	3200

p_view

emp_no	emp_name	salary
10	Haas	52750
20	Thompson	41250
30	Kwan	40175
50	Geyer	38250
60	Stern	36250
70	Pulaski	36170
80	Henderson	35750
110	Lucchessi	38170
130	Quintana	33800
140	Nicholls	35420
150	Adamson	30280
170	Yoshimura	28680
190	Walker	20450
230	Jefferson	22180
240	Marino	33760
290	Parker	15340
300	Smith	
310	Setright	15900

rp_view

emp_no	emp_name	salary
60	Stern	36250
70	Pulaski	36170
150	Adamson	30280
170	Yoshimura	28680
190	Walker	20450
230	Jefferson	22180
240	Marino	33760
290	Parker	15340
300	Smith	
310	Setright	15900

Group Fields and Group Functions

The purpose of group fields is to *partition*, or *categorize*, or *group* the rows of a table by equal values in a selected column or equal sets of values in a set of columns, or by values within specified intervals in a column or set of columns. For example, consider the `dept_no` and `sex` columns in the `employees` table in the base table figure.

The two columns on the left in the partitioning table show the distinct entries in the `dept_no` column and the row indices for the employees in each department. For instance, the employees in department D11 are in rows 4, 10, 11, and 12 of the `employees` table. Thus the second column represents the partitioning when the column `dept_no` is the group field.

The two columns on the right in this table represent the partitioning when the columns `dept_no` and `sex` are the set of group fields. When `dept_no` alone is the group field there are eight groups in the partition because there are eight department numbers. When `dept_no` and

sex are the set of group fields there are sixteen groups, consisting of the females in each of the eight departments, and the males in each department.

Partitioning of the employees Table Based on One and Two Group Fields

dept_no	One Group Field: dept_no	Two Group Fields: dept_no and sex	
	dept_no row indices	dept_no and female	dept_no and male
A00	0, 7	0	7
B01	1		1
C01	2, 8, 9	2, 8, 9	
E01	3		3
D11	4, 10, 11, 12		4, 10, 11, 12
D21	5, 13, 14	5	13, 14
E11	6	6	
D31	15, 16, 17	17	15, 16

The function *t.group* is used to establish views based on group fields. In the example of *dept_no* as a group field for the *employees* table, a view could be formed as follows:

```
`employees `bd_view t.group `dept_no
```

As with *t.also*, the effect of *t.group* is to create the view called *bd_view*, but the view must still be populated using *t.send*. If the *salary* column is sent from the *employees* table to *bd_view*, it is not unreasonable to expect that the items of the target column would be salaries grouped by department. Typically, however, it is not these groups that one wants, but the result of some function applied to the groups. For instance, one might want the largest value in each group, the smallest, or the average. A function applied to the groups is called a *group function*, or *summarization function*. For instance, to get the largest salary in each group, the group function would be specified as follows:

```
`employees `bd_view t.send (`salary;⌈/)
```

Different columns can be sent with different group functions. For instance, if the *emp_no* field is sent to *bd_view* and its items are grouped by departments, then its group function might simply collect all the employee numbers in each group. This group function would be specified as follows:

```
`employees `bd_view t.send (`emp_no;<)
```

For more on group functions, see "How Group Functions are Specified and How t Applies Them".

How t Manages Group Fields

Exactly how are group fields maintained? Executing `t.group` establishes a mapping from the row indices of the source table to the row indices of the target table, which in this example is from the `employees` table to the `bd_view` table, and is:

```
      (<@1 (D employees.dept_no) OE employees.dept_no
       )/¨<⍳#employees.dept_no
<   0 7
<   1
<   2 8 9
<   3
<   4 10 11 12
<   5 13 14
<   6
<   15 16 17
```

The function `D{x}:((x⍳x)=⍳#x)/x` is the usual one that produces the distinct items of the `dept_no` column. The function `OE{x;y}:x≡@1 1 0 y` generalizes the outer product `a∘.=b` to apply to items of matrices instead of items of vectors. The result of the above expression is a vector of enclosed index vectors that reflects the grouping defined by the group field; compare this result with the table.

For example, when the `salary` column is sent to the `bd_view` table, the fifth element of the mapped column, `bd_view.salary[4]`, is computed from elements 4, 10, 11, and 12 of the source column, and is `⌈/employees.salary[4 10 11 12]`, or 36250. The group function `⌈/` was specified when the `salary` column was sent to departments. Different functions can be specified for different columns.

This mapping is maintained by t in the table dependency `bd_view._J`. When the `salary` column is sent from `employees` to `bd_view` with the group function `⌈/`, the new column is a dependency on the source column whose definition is essentially:

```
bd_view.salary:>⌈/¨bd_view._J#¨<employees.salary
```

(but see the use of `_index` in the definition of `col` in "How t Manages Row and Column Selection").

There are two parts to this dependency:

1. `bd_view._J` is a partition of the row indices of the employees table with one element for each row in the `bd_view` table. When a column is sent from employees to departments, say salary, that column is first partitioned according to this variable, essentially as:
   `partition_of_salary←bd_view._J#¨<employees.salary`
2. The partitioned column is a vector of subcolumns, with no item of the column appearing in more than one subcolumn. The group function is applied to each of these subcolumns to produce the target column in the `departments` table:
   `⌈/¨partition_of_salary`

Grouping by Intervals

So far in this chapter, items in group fields have been grouped by equality. An important variation of this practice, particularly for numeric group fields, is to group items according to intervals of minimum and maximum values, i.e., to group them in ranges. For example, in the `employees` table in the base table figure, it is generally more useful to group employees by

salary ranges than by identical salaries. For instance, the question as to how many salaries fall within each $5000 increment can be answered as follows. First, make a view of the *employees* table with a few representative fields including *salary*, and in that view form two new columns representing the lower limit and upper limit of the salary interval for each salary. These new columns will calibrate the group field.

```
`employees `e_view t.only ()
`employees `e_view t.send `emp_no `emp_name `salary
`e_view t.define (`min_range;
                  '⍎1+5000×⌊employees.salary÷5000')
`e_view t.define (`max_range;
                  '⍎5000×⌈employees.salary÷5000')
```

Now form a second view based on salary as a group field with intervals of $5000:

```
`e_view `i_view t.group (`salary;5000×⍳12)
```

Finally, send salary to the second view with the A+ primitive # as its group function, and send both *min_range* and *max_range* with the default group function ("first item"):

```
`e_view `i_view t.send (`min_range;;
                        `max_range;;
                        `salary_count;(`salary;#))
```

Both views are shown in the grouping figure. The reason for creating the *min_range* and *max_range* columns in the *e_view* table and sending them to *i_view* should now be apparent: in *i_view*, they serve to calibrate the *salary_count* table. In a real application, they would most likely not appear in *e_view*.

Furthermore, in a real application the last row of the i_view table, which is due to the missing salary for Smith, would be filtered out.

Grouping By Intervals On the Salary Column Of e-view:

e_view

emp_no	emp_name	min_range	max_range	salary
10	Haas	50001	55000	52750
20	Thompson	40001	45000	41250
30	Kwan	40001	45000	40175
50	Geyer	35001	40000	38250
60	Stern	35001	40000	36250
70	Pulaski	35001	40000	36170
90	Henderson	35001	40000	35750
110	Lucchessi	35001	40000	38170
130	Quintana	30001	35000	33800
140	Nicholls	35001	40000	35420
150	Adamson	30001	35000	30280
170	Yoshimura	25001	30000	28680
190	Walker	20001	25000	20450
230	Jefferson	20001	25000	22180
240	Marino	30001	35000	33760
290	Parker	15001	20000	15340
300	Smith			
310	Setright	15001	20000	15900

i_view

min_range	max_range	salary_count
50001	55000	1
40001	45000	2
35001	40000	6
30001	35000	3
25001	30000	1
20001	25000	2
15001	20000	2
		1

Static Grouping

There are useful variations of the group fields called direct and indirect static group fields, which have these characteristics:

- The map from the source table to the target table can be computed once and stored. This is useful for large, relatively fixed files for which these computations are expensive.
- The map is represented as a triple of simple integers instead of a nested vector of integer vectors, which is better for large files and easy to save in mapped files for later use.

To illustrate the indirect static link, suppose that the group view above had been formed with `t.group_i` instead of `t.group`:

```
      `employees `bd_view t.group_i `dept_no
```

Let `x` be the mapping `bd_view._J` defined above for the `t.group` case. Then the mapping `bd_view._J` in the `t.group_i` case is simply a vector of counts of the items of the `x`, namely `⊃#⌸x`:

```
      bd_view._J
 2 1 3 0 1 4 3 3 1 0
```

In addition, `t.group_i` generates a permutation vector named `employees._H_bd_view` that sorts the group field `dept_no` into nondecreasing order, and also a vector named `bd_view._K` of first occurrences of the unique elements of the group field in the sorted group field:

```
     employees._H_bd_view
 0  7  1  2  8  9  4 10 11 12  5 13 14 15 16 17  3  6
     bd_view._K
 0  2  3 18 16  6 10 13 17 18
```

Then for any n, the n-th item of the mapping `x` defined above can be obtained from these three vectors:

```
employees._H_bd_view[bd_view._K[n]+⍳bd_view._J[n]
```

Consequently, the group mappings for `t.group` and `t.group_i` are functionally equivalent.

Once these variables have been created, they can be saved in mapped files, and in such a way that they will be retrieved automatically by t when needed. Suppose all the columns of the employees table are kept in a directory named `source`, and nothing else is kept there. Save `employees._H_bd_view` in that same directory, and with the file name the same as the variable name:

```
'source/_H_bd_view.m'⍠employees._H_bd_view
```

The target table is a view and therefore its columns did not come directly from mapped files. So, choose a directory that will only hold the saved variables `bd_view._J` and `bd_view._K`, say `target`:

```
'target/_J.m'⍠bd_view._J
'target/_K.m'⍠bd_view._K
```

In a later session, the two tables are opened as follows:

```
(`employees;'source') t.open ()
(`bd_view;'target') t.open ()
```

- yes, even the view is explicitly opened, in order to retrieve the saved table variables - and `bd_view` is made a view of employees:

```
`employees `bd_view t.only ()
```

Send the grouping fields:

```
`employees `bd_view t.send (`dept_no;)
```

The state is now the same as that after the original evaluation of `t.group_i`. For example, send the `salary` field to the view, as in the `t.group` example (note that this can be combined with sending the grouping fields):

```
`employees `bd_view t.send (`salary;⌈/)
```

In addition to the indirect static group, there is a direct static group denoted by `t.group_d`. The difference is that the permutation is not computed, i.e., in the above example, `employees._H_bd_view`. It is still needed, however, and must be supplied by the user, similarly to `t.break_b`. For example, after the following is executed:

```
`employees `bd_view t.break_d `dept_no
```

the permutation vector must be defined before any fields are sent to the view.

Successive Multiple Group Fields

The partitioning table illustrates grouping on multiple fields, in particular:

```
`employees `view t.group `dept_no `sex
```

As that table indicates, rows of `employees` are grouped together based on equal rows of the column pair `dept_no` and `sex`. An interesting variant of grouping on multiple fields is provided by `t.report`. If the following is executed:

```
`employees `view t.report `dept_no `sex
```

the result is as if an intermediate table were formed that grouped the `employees` table according to the first column named in the right argument, namely `dept_no`, and then view were formed from that intermediate table by further classifying the members of each department by `sex`, i.e. the second column named in the right argument. In fact, that intermediate table is actually formed. If more columns had been named in the right argument then this successive refinement method would have continued for every named column, in the order of the names in the list, and an intermediate table for every step except the last would be created.

A special first column named `view._` is automatically generated, providing row labels for the successive refinements. If the screen management system called s is present, then the following provides a display of a representative examples of `t.report`:

```
`employees `view t.send (`salary;⌈/)
show t.table `view
```

Linking Tables

Linking is the basic operation for relating the contents of two tables. In the simplest cases the two tables must have a common field, called the *link field*; more general cases with several link fields will come up later. That is, there is a column in one table with the same name as a column in the other, and these two columns are in the same domain (see "Table Row and Column Domains"). In most practical cases, the two columns have some matching items but are not identical. Linking is not a symmetric operation. One table is selected as the *partition side* of the link, largely because its link field has distinct items, and the other then becomes the *replication side*. The link field on the partition side is also called the *index field*, and the one on the replication side the *grouping field*, or *group field*.

A link is a relationship between two tables, based on identical items in their common link field. The relationship is expressed in terms of a correspondence between the row indices of the partition side and the row indices of the link side. The correspondence is illustrated in the next table for the two tables in the base table figure and the common field `dept_no`. For example, department number C01 appears in row 2 of the `departments` table and rows 2, 8, and 9 of the `employees` table. The asymmetry of the link is a reflection of the asymmetry of this correspondence, which is one to many in one direction (and many to one in the other).

Once a link is established, columns can be sent, or mapped, from either side of the link to the other based on the correspondence between the row indices of the two tables. For example, suppose the `dept_name` column is mapped from the `departments` table to the `employees` table to form a new `dept_name` column. Then item 2 of the `dept_name` column in

departments, "Information Center", will appear in rows 2, 8, and 9 of the new *dept_name* column. Or, if the *salary* column in the *employees* table is mapped to the *departments* table to form a new *salary* column there, then the salaries of the employees Pulaski, Jefferson, and Marino (i.e., rows 5, 13, and 14 of the *employees* table) will all appear in row 6 of the new *salary* column.

Note that the mapping from the replication side of the link to the partition side is essentially the same as the one from a source table to a view created by *t.group*, up to a permutation of target table.

Row Index Correspondence of departments and employees based on dept_no

Row Index in departments	dept_no items	Row Index in employees
0	A00	0, 7
1	B01	1
2	C01	2, 8, 9
3	D01	
5	D11	4, 10, 11, 12
6	D21	5, 13, 14
7	D31	15, 16, 17
4	E01	3
8	E11	6
9	E21	

Example

The most common situation in linking two tables occurs when the items of one of the link fields are distinct, while those of the other may or may not be distinct. In this case the table whose link field is known to have distinct items should be selected as the partition side of the link, and the other becomes the replication side. As an example, consider once again the two tables in the base table figure.

These tables have a common field called *dept_no*. If they are linked on that field (as in the table above), tables corresponding to the results of the following queries (among many) can then be built:

- Query 1: what are the department names for all employees?
- Query 2: what is the maximum salary in each department?

Since department numbers uniquely identify departments, the *departments* table is chosen to be the partition side of the link. The link is established as follows:

```
`departments `employees t.link `dept_no
```

The left argument of *t.link* is always a two-element symbolic vector whose first element identifies the partition side of the link, and whose second element identifies the replication side. The right argument is (a list of) the link field(s). Once the link is established, any number of columns can be sent from one table to the other in order to answer a variety of queries. To answer the first query, send the *dept_name* column from the *departments* table to the *employees* table, as follows:

```
`departments `employees t.send `dept_name
```

The answer is then contained in the columns *emp_name* and *dept_name* of the augmented *employees* table. These columns cannot be displayed separately because the column *emp_name* already appears in the display of *employees* (see the base table figure) and the screen management system does not allow an object to appear on the screen more than once. Therefore, for display purposes, create and populate a new table, *query1*, as follows:

```
`employees `query1 t.only ()
`employees `query1 t.send `emp_name `dept_name
```

See the query tables figure for the display of this table.

As for the second query, send the *salary* field in the *employees* table to the *departments* table, as follows:

```
`employees `departments t.send (`salary;⌈/)
```

(the group function ⌈/ implements the "maximum salary" part of the query). Create a query table as follows (see the query tables figure):

```
`departments `query2 t.only ()
`departments `query2 t.send `dept_name `salary
```

Sample Query Tables Created Via *t.link*:

query1

emp_name	dept_name
Haas	Spiffy Computer Service Div.
Thompson	Planning
Kwan	Information Center
Geyer	Support Services
Stern	Manufacturing Systems
Pulaski	Administration Systems
Henderson	Operations
Lucchesi	Spiffy Computer Service Div.
Quintana	Information Center
Nicholls	Information Center
Adamson	Manufacturing Systems
Yoshimura	Manufacturing Systems
Walker	Manufacturing Systems
Jefferson	Administration Systems
Marino	Administration Systems
Parker	Order Processing Systems
Smith	Order Processing Systems
Setright	Order Processing Systems

query2

dept_name	sal
Spiffy Computer Service Div.	52750
Planning	41250
Information Center	40175
Development Center	
Support Services	38250
Manufacturing Systems	36250
Administration Systems	36170
Order Processing Systems	15340
Operations	35750
Software Support	

How t Manages Table Links

Exactly how are table links accomplished? Executing *t.link* establishes two mappings between the two sides of the link, represented in the above table as correspondences between the row indices in column 1 and the row indices in column 3. One is a mapping from the rows of the *departments* table to the rows of the *employees* table, and in general from the partition side to the replication side, which is:

```
     departments.dept_no↓employees.dept_no
 0 1 2 4 5 6 8 0 2 2 5 5 5 6 6 7 7 7
```

E.g., where the *dept_name* column is sent to the *employees* table, item *employees.dept_name*[6] in the mapped column equals item *departments.dept_name*[8] in the source.

This mapping is maintained by t in the table dependency *employees._I_departments*. When the column *dept_name* is sent from *departments* to *employees*, the new column is a dependency on the source column whose definition is essentially:

```
employees.dept_no:{
          employees._I_departments#departments.dept_no}
```

(but see the use of *_index* in the definition of *col* in "How t Manages Row and Column Selection").

A second mapping in the other direction is also established, i.e., from the replication side to the partition side, which is similar to the mapping associated with group fields (see "How t Manages Group Fields"). In this example the mapping is from the *employees* table to the *departments* table, and is:

```
      (<@1 departments.dept_no 0E employees.dept_no
       )/¨<⍳#employees.dept_no
<   0 7
<   1
<   2 8 9
<
<   3
<   4 10 11 12
<   5 13 14
<   15 16 17
<   6
<
```

The function *0E* is defined in "How t Manages Group Fields". Compare this result with the correspondence table, and the group field mapping in "How t Manages Group Fields".

For example, when the *salary* column is sent to the *departments* table, the sixth element of the mapped column, *departments.salary*[5], is computed from elements 4, 10, 11, and 12 of the source column, and is ⌈/*employees.salary*[4 10 11 12], or 36250. The group function ⌈/ was specified when the *salary* column was sent to *departments*. Different functions can be specified for different columns.

This mapping is maintained by t in the table dependency *departments._J_employees*. When the column *salary* is sent from *employees* to *departments* with the group function ⌈/, the new column is a dependency on the source column whose definition is essentially:

```
departments.salary:{
       >⌈/¨departments._J_employees#¨<employees.salary}
```

(but see the use of *_index* in the definition of *col* in "How t Manages Row and Column Selection").

It is worth noting that the name of each map from one partner in a link to the other contains the name of the partner. This suggests that a table can participate in more than one link, since there would be no name conflicts in the table dependencies by which the multiple links would be managed. This is in fact true. Moreover, a table participating in several links can serve as the partition side in some and the replication side in others.

Multiple Group Fields

How can effective links be done if no column can be guaranteed to have distinct items? The answer is that link fields are not restricted to a single column; any set of columns will do, as long as columns with these names appear in both tables, and the two sets of columns are in the same domain (see "Table Row and Column Domains"). The partition side of the link is then the table in which the rows have distinct sets of entries in the set of link fields. The mapping from the partition side of the link to the replication side resembles the b context function *b.p* more than dyadic ⍳, but is more general, and the mapping in the other direction is comparably general. Several examples of linking on more than one column appear in the "The Relational Set Operations: Union, Intersection, and Difference".

Duplicate Items in a Link Field on the Partition Side

Even if the items of a link field or the rows of link fields are not distinct on the partition side of a link, the link can still be made; how effective it will be depends on the application. The effect of partition side duplicates can be seen from the mappings between the two sides of the link. When there is one link field, the mapping from the partition side to the replication side is of the form:

```
      part.link_field ⍳ repl.link_field
```

If the *i*th and *j*th items of the column *partition.link_field* are identical and *i* is less than *j*, then *j* does not appear in this mapping, i.e., *j* is not in the result of the above expression. Consequently the *j*th item of any column sent to the replication side of the link does not appear in that table. As for the mapping in the other direction, which in its simplest form is

```
(<@⍳ part.link_field ∘.= repl.link_field
 )/¨<⍳#repl.link_field ,
```

the *i*th and *j*th elements created by this mapping are identical, and so the *i*th and *j*th items of any column sent to the partition side are identical.

Static Links

Analogous to *t.group*, there is an indirect static link function *t.link_i* and a direct static link function *t.link_d*. They produce variations of the map from the replication side to the partition side which are the same as the variations produced by *t.group_i* and *t.group_d*. In particular, *t.link_d* does not produce the required permutation vector on the replication side; this vector must be defined by the user before any columns are sent across the link.

There is one difference between static links and static groups. After the link is established and saved, and after the two linked tables are opened in a later A+ session, link pointers must be manually set. For example, if the static link had been originally established as:

```
`departments `employees t.link_i `dept_no
```

then the following establishes the link pointers in a later session:

```
employees._L[,]←`departments
departments.!R[,]←`employees
```

The Relational Set Operations: Union, Intersection, and Difference

The relational set operations provide examples of linking across multiple fields. They apply to pairs of tables with the same number of columns, where for every column index i, the i-th columns of the two tables are in the same domain (see "Table Row and Column Domains"). In other words, it is possible to link the two tables across the full set of columns. To reiterate the point made earlier in the chapter, it is assumed that the tables have distinct rows (which is one of the criteria in the definition of relational tables).

Tables For the Relational Set Operation Examples:

salary

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
10	Haas	A00	650101	f	52750	
20	Thompson	B01	731010	m	41250	
30	Kwan	C01	750405	f	40175	
50	Geyer	E01	490817	m	38250	
60	Stern	D11	730914	m	36250	
70	Pulaski	D21	800930	f	36170	
90	Henderson	E11	700815	f	35750	
110	Lucchessi	A00	580516	m	38170	
130	Quintana	C01	710728	f	33800	
140	Nicholls	C01	761215	f	35420	
150	Adamson	D11	720212	m	30280	
170	Yoshimura	D11	780915	m	28680	
190	Walker	D11	740726	m	20450	
230	Jefferson	D21	661121	m	22180	
240	Marino	D21	791205	m	33760	
280	Parker	D31	800530	m	15340	4780
310	Setright	D31	640912	f	15900	3200

commission

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
280	Parker	D31	800530	m	15340	4780
300	Smith	D31	720618	m		17750
310	Setright	D31	640912	f	15900	3200

Suppose that instead of the *employees* table, there are two other base tables, a *salary* table for all employees who receive a salary, and a *commission* table for all employees who receive a commission. For the purpose of illustration, the *salary* table can be built from the *employees* table as follows:

```
`employees `salary t.only 'salary≠`int⊃t.NA'
`employees `salary t.send ()
```

(Note that there is now a *salary* table and a *salary* column.) There is one employee who receives no salary. The missing salary information for that employee is represented in the *salary* column by an NA value. That value was chosen to be the default numeric NA value, *`int⊃t.NA*, when the mapped files for the base tables were built. Consequently the selection expression in the right argument of *t.only* above selects all employees who receive a salary. The resulting *salary* table will the same as the *employees* table, except that the row with emp_no equal to 300 is missing (see the base tables figure, and the operations figure). The *commission* table is formed in the same way, and contains the last three rows of the *employees* table.

```
`employees `commission t.only 'commission≠`int⊃t.NA'
`employees `commission t.send ()
```

See the operations figure.

Link these tables on the full set of columns; since the rows of either table are distinct, either table can be used on the partition side of the link:

```
`salary `commission t.link salary._T
```

Intersection

If a row of the commission table occurs in the salary table, then the corresponding element of the mapping

```
commission._I_salary
```

must be a valid row index of the *salary* table, and therefore not equal to the t-created variable *salary*._*N*, whose value is the number of rows in *salary*. By definition, the set of rows of the *commission* table that also appear in the *salary* table is the intersection of the two tables, and therefore the intersection table can be defined as:

```
t.only{`commission `intersection;
       'commission._I_salary≠salary._N'}
`commission `intersection t.send ()
```

Examples of The Set Operations:

intersection

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
290	Parker	D31	800530	m	15340	4780
310	Setright	D31	640912	f	15900	3200

comm_minus_sal

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
300	Smith	D31	720619	m		17750

union

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
10	Haas	A00	650101	f	52750	
20	Thompson	B01	731010	m	41250	
30	Kwan	C01	750405	f	40175	
50	Geyer	E01	490817	m	38250	
60	Stern	D11	730914	m	36250	
70	Pulaski	D21	800930	f	36170	
90	Henderson	E11	700815	f	35750	
110	Lucchessi	A00	580516	m	38170	
130	Quintana	C01	710728	f	33800	
140	Nicholls	C01	761215	f	35420	
150	Adamson	D11	720212	m	30280	
170	Yoshimura	D11	780915	m	28680	
190	Walker	D11	740726	m	20450	
230	Jefferson	D21	661121	m	22180	
240	Marino	D21	781205	m	33760	
290	Parker	D31	800530	m	15340	4780
310	Setright	D31	640912	f	15900	3200
300	Smith	D31	720619	m		17750

sal_minus_comm

emp_no	emp_name	dept_no	hire_date	sex	salary	commission
10	Haas	A00	650101	f	52750	
20	Thompson	B01	731010	m	41250	
30	Kwan	C01	750405	f	40175	
50	Geyer	E01	490817	m	38250	
60	Stern	D11	730914	m	36250	
70	Pulaski	D21	800930	f	36170	
90	Henderson	E11	700815	f	35750	
110	Lucchessi	A00	580516	m	38170	
130	Quintana	C01	710728	f	33800	
140	Nicholls	C01	761215	f	35420	
150	Adamson	D11	720212	m	30280	
170	Yoshimura	D11	780915	m	28680	
190	Walker	D11	740726	m	20450	
230	Jefferson	D21	661121	m	22180	
240	Marino	D21	781205	m	33760	

Set Difference

The set difference "`commission` minus `salary`" is the set of rows of the `commission` table that are not in the `salary` table, i.e., the complement of the intersection relative to `commission`. The difference can be formed simply by changing ≠ to = in the right argument of `t.only` above, or by using the same right argument in the complementary t function called `t.not`:

```
t.not{`commission `comm_minus_sal;
      'commission._I_salary≠salary._N'}
`commission `comm_minus_sal t.send ()
```

See the examples figure.

The other set difference, "*salary* minus *commission*", which is the set of rows of *salary* that are not in *commission*, can be obtained from the partition map *salary._J_commission*; it consists of all rows *j* of *salary* for which the *j*th partition is empty, i.e.,

```
t.only{`salary `sal_minus_comm;
       '~>#¨salary._J_commission'}
`salary `sal_minus_comm t.send ()
```

See the examples figure.

Union

The union of the *salary* and *commission* tables is the catenation of *salary* and the set difference table *comm_minus_sal* formed above, rows on rows:

```
`salary `comm_minus_sal `union t.cat ()
`salary `comm_minus_sal `union t.send ()
```

See the examples figure. The resulting table, named *union*, is the same as *employees*, except that the row for *emp_no* equal to 300 is in a different position. To make *union* the same as *employees*, sort it on the *emp_no* column:

```
`union t.sort `emp_no
```

Outer Union

The set operations are too restrictive to be generally useful because they only apply to pairs of tables that can be linked on their entire set of columns, a situation that does not arise naturally in applications very often. For example, the *salary* and *commission* tables in the previous example would most likely not occur in a real application; instead, the *salary* table would not have the *commission* column and the *commission* table would not have the *salary* column. The outer union is the relational operation that produces the *employees* table from the more realistic salary and commission tables, which are defined below as *salary*1 and *commission*1:

```
`employees `salary1 t.only 'salary≠`int⊃t.NA'
`employees `salary1 t.send (
   employees._T≠`commission)/employees._T
`employees `commission1 t.only (
   'commission≠`int⊃t.NA')
`employees `commission1 t.send (
   employees._T≠`salary)/employees._T
```

This example is reduced to the previous one - and in general outer union is reduced to union - by constructing the *salary* and *commission* tables from *salary*1 and *commission*1. To do that, link *salary*1 and *commission*1 on all columns except *salary* and *commission*:

```
`salary1 `commission1 t.link (
   `salary1._T≠`salary)/salary1._T
```

and send each one the missing column from the other:

```
`salary1 `commission1 t.send `salary
`commission1 `salary1 t.send (`commission;)
```

The function to be applied to partitions when generating the *commission* column in *salary1* is "first element", as indicated by the right argument to *t.send* in the last expression (see "How Group Functions are Specified and How t Applies Them"). Since each partition happens to consist of a single element in this case (the collections of link fields in the two tables are identical), "first element" simply selects that element. The resulting *salary1* and *commission1* tables are identical to *salary* and *commission*.

Definitions of t-Context Functions

Selectors

In every example above of *t.only* and *t.also*, the right argument defining the selection, when not the Null, is a character vector containing a valid A+ expression. One example in "Row and Column Selection", suggests the use of *t.in* to permit special syntax, namely a right argument of the form *"a,b,c"* in place of the typical right argument to Member of (∈) of the form *3 1⍴"abc"*. It is possible to extend this technique with application-specific functions that play the same role as *t.in*. It is also possible to permit more Englishlike selections on specific columns by connecting application-supplied selection evaluation functions to the t selection functions. These selection evaluation functions are called *selectors*.

For example, it is possible to permit selections on date fields of the form *'1/1/93 to 10/1/93'* if there is an application-supplied selector function that can parse this expression and apply it to table columns. A selector function must satisfy the following conditions:

- the syntax of a selector function is the same as callback functions, namely: *selector{s;d;i;p;c;v}*
  (the index argument *i* holds the selection criteria, such as *'1/1/93 to 10/1/93'*, the context argument *c* holds the table name, the variable argument *v* holds the column name, and the data argument *d* holds the value of the column variable);
- the result of a selector function is a boolean vector of the same length as the table column named by the arguments *c* and *v*, with 1 indicating that the corresponding item of the column is selected.[1]

For instance, if date ranges are given as *'1/1/93 to 10/1/93'*, the selector function might be:

```
date_selection{s;d;i;p;c;v}:{
      i←' ',i;
      b←' '=i;
      (f;t)←⍸¨0 2#(⊂b)⊂i;
      (d≥f)∧(f≤t)
}
```

A selector function can be connected to the column of a table in one of two ways. It may be specified when a table is opened with *t.open*, or the t.selector attribute is specified for the column with the selector function as its value, as in:

```
`prices.date_col _set (`t.selector;date_selection)
```

where *prices* is the table containing *date_col*. Once connected, whenever a selection of the following form is made on the *date_col* column of the table named *prices*:

```
(`date_col;'1/1/93 to 10/1/93')
```

the selector function is called. This type of selection can be made with any of the t selection functions, `t.only`, `t.also`, and `t.not`.

A selector can also be specified by a pair `(fcn;static_data)`, as in

```
`prices.date_col _set (`t.selector;
                       (date_selection;date_style))
```

where the data has the same significance as in the right argument of `_scb`.

Several column selections can be made at once, as in:

```
(`date_col;'1/1/93 to 10/1/93';
 `time_col;'9:00 A.M. to 4:30 P.M.')
```

Each of the two selector functions will create a boolean vector. The two vectors can be combined into a single selection result in one of two ways, either by ORing or ANDing them together (although in this particular example only AND seems to make sense). The function to be used, ∨ or ∧, is the contents of the t-variable `t.CONNECT`.

A column in a target table that is the image of one in a source table will use the selector of its source column if one of its own has not been specified.

Selectors are specified in the right arguments of `t.open` and `t.define`.

Dependency Frames

A dependency frame is a character vector whose contents are an A+ expression preceded by a colon.

When a column is sent from one table to another, by default the target column is defined to be a dependency on the source column, basically of the following form, where `f` is defined by t:

```
target.col:f{source.col}
```

A dependency frame is the means of modifying this definition (see `t.send`). In this example, wherever the column name `col` appears in the dependency frame, it is replaced by `f{source.col}`, and the resulting expression becomes the definition of the dependency `target.col`. For example, if the dependency frame is `":⍵col"`, the target column dependency will be:

```
target.col:⍵f{source.col}
```

Dependencies are specified in the right argument of `t.send`.

How Group Functions are Specified and How t Applies Them

There are four ways that t applies group functions to a partitioned column, and the method chosen in most cases depends on how the group function is denoted in the right argument to `t.send`. There are various ways to attach a group function to a column, the simplest being as a column-function pair such as `(salary;f)`; see the definition of `t.send` for the others. No matter how the attachment is done, the ways in which group functions themselves are denoted are always the same.

Using the partitioned column `partition_of_salary` defined in "How t Manages Group Fields", the ways t applies group functions are:

1. if the group function is specified by <, it is, in effect, not applied. The new salary column in the bd_view table is:
   ```
   partition_of_salary
   ```
2. if the group function is specified by the Null then it is taken to be the first item function
   ```
   fe{v}:if (0=#v) _N else 0#v
   ```
   which is formally applied in the standard way (see 3) below):
   ```
   >fe¨partition_of_salary
   ```
   However, t maintains the t-created variable `bd_view._K` that is defined to be the vector of first items of the partition subcolumns:
   ```
   bd_view._K←>fe¨ bd_view._J
   ```
   and consequently the new salary column in the departments table is:
   ```
   bd_view._K#employees.salary
   ```
3. if the group function is specified by any valid function expression `f`, e.g. + or +/, it is applied as follows:
   ```
   >f¨partition_of_salary
   ```
4. if the group function is specified as a character vector containing any valid function expression, as in `"f"`, it is applied as follows:
   ```
   f¨partition_of_salary
   ```

Similar definitions apply to group functions for linked tables.

Business Days `t.calendar{s}`

Argument and Result

The argument `s` is a character vector, and the result is an integer vector.

Definition

The argument `s` specifies the beginning date, ending date of a time series in the form:

```
'begin-date → end-date'
```

where both beginning date and ending date are of the form mm/dd/yyyy. The result is an integer vector of all business days between beginning date and ending date. Each integer in the result is of the form

```
0 100 100 ⊥year,month,day
```

where `year` is a four-digit integer, as in 1993, and `month` and `day` are at most two-digit integers.

Note that the script `dio.+` must be loaded before this function is used.

Calendar Indices `t.series{w;x}`

Arguments and Result

The left argument `w` is a symbol scalar holding the name of a global variable, the right argument `x` is a character vector, and the result is an integer vector.

Definition

It is assumed that the variable named in the left argument has had its `t.calendar attribute set, either indirectly with t.open or directly with _set. If so, this function produces indices into the value of that attribute, as follows (where calendar is w _get `t.calendar):

```
w t.series 'begin_date → end_date'
     ⍳ for begin_date to end_date

w t.series '           → end_date'
     ⍳ for 0 to end_date

w t.series 'begin_date →         '
     ⍳ for begin_date to (#calendar)-1

w t.series '           →         '
     ⍳ for ⍳#calendar
```

The extra spaces in the right arguments are only for readability. Both begin_date and end_date are in the same format as the right argument to t.calendar, basically dd/mm/yyyy.

Catenation t.cat{w;}

Argument and Result

The argument w is a symbol vector which must have at least three elements to be meaningful. The result equals w.

Definition

Suppose that w is `s0 `s1 ... `sn `v. The effect of this function is to initialize the view `v as the catenation of the tables `s0 through `sn. Every element `si is the name of an existing table, either a base table or a view. The names of the columns in any `si must be the same as in any other `sj. In addition, any pair of columns from `si and `sj with the same name must be in the same domain (see "Table Row and Column Domains").

When t.send is subsequently executed to populate the view `v with columns, its left argument need not equal w, even though w as the right argument of t.send is by far the most useful case. Only columns of those tables specified in the left argument to t.send will appear in `v, and in the same order (top to bottom) as table names appear in that left argument (left to right).

Example

See the set union example in "The Relational Set Operations: Union, Intersection, and Difference".

Close Columns of a Table t.close{w;f}

Arguments and Result

The left argument w is a symbol scalar, while the right argument f and the result are symbol vectors. The right argument f can also be the Null.

Definition

The left argument *w* is the name of a t-table. If the right argument *f* is nonempty, then its elements are the names of columns of the t-table *w*. In this case the following is done:

- the columns named in *f* are removed from the table *w*, i.e., from *w*%`*_T*;
- if the table *w* is a view, the objects that define the columns named in *f* are removed from the table context, but not if *w* is a base table;
- the columns in any other t-tables that were sent from any columns of *w* named in *f* are removed from those tables, and the objects that define them are removed from those table contexts; in addition, those table contexts are removed for views that now have no columns;
- any links to other t-tables are removed in which a column named in *f* is a link field.

If *f* is the Null then all columns of the table are treated in this manner. In addition, all table variables are removed, and in the case of a view, the table context itself is removed.

The result is a list of names of all t-tables whose table variables *_T* are modified by this evaluation.

Note that if the right argument is the Null and the table is bound to a display class when closed, it is unbound.

Complement *t.not{w;e}*

Arguments and Result

The left argument *w* is either a two-element symbol vector, a one-element symbol vector, or a symbol scalar. The right argument *e* is either a character or integral vector or scalar, or the Null. The result is an integer scalar.

Definition

This function is the complement of *t.only*; it behaves exactly like *t.only*, except that it selects those rows that would not be selected if *t.only* were executed with the same arguments. If both the source and target are named in the left argument, then *complement* refers to the source table, while if only the target is named, *complement* refers to current view. In particular, if the right argument is the Null and both the source and target are named in the left argument, the resulting view has no rows, while if only the target is named, the resulting view has all rows in the source that are not in the current view.

Example

See the example in "Set Difference", as well as the example under *t.only* for a numeric right argument. Also:

```
     tab.a←⍳100
     `tab t.open `a
     `tab `view t.only 'a<90'
90
     `tab `view t.send `a
     `view t.only 'a<40'
40
     `view t.not 'a<10'      ⍝ The complement in the view.
```

```
30

    `tab `view t.not 'a<10' ⍝ The complement in the source.
90
    `view t.not ()   ⍝ The complement of the current view in the source.
10
    `tab `view t.not ()     ⍝ The empty view.
0
```

Create and Initialize a Base Table *t.open{w;x}*

Arguments and Result

The left argument *w* is either a symbol scalar, a character vector, a two-element nested vector consisting of a symbol scalar and a character vector in either order, or the Null. The right argument *x* is a symbol vector, an association list, a nested association list, a slotfiller, or a nested slotfiller. The result is a symbol scalar.

Definition

This function creates and initializes t-tables. The meaning of the left argument is:

1. A symbol scalar left argument is the name of the table to be opened. Any mapped files designated to be opened by the right argument are assumed to be in the current directory (./, i.e., the directory displayed when *$pwd* is executed in the A+ session).
2. A character vector left argument is the name of the directory location of the mapped files in the table. The empty character vector designates the current directory (see the first case). The name of the table is taken to be that of the current context.
3. The meanings of the symbol scalar and character vector in the two-element nested vector form of the left argument are the same as 1 and 2.
4. The Null left argument means that the name of the table is taken to be that of the current context, and the directory is the root directory (./).

The meaning of the right argument is:

- A symbol in a symbol vector right argument is both the name of a column of the table being opened and the name of the object defining that column, according to the left argument, as follows:
    - If the left argument is a symbol, then the right argument consists of names of existing A+ objects. An unqualified name is assumed to refer to an object in the context of the table being opened.
    - For any other left argument, the right argument consists of names of files in the directory determined by the above left argument rules.
- A Null right argument means:
    - If the left argument is a symbol, then the table columns are of all global existing variables and dependencies in the context named by the left argument.
    - Any other left argument specifies a directory. If that directory contains a file named .T, that file is assumed to be a mapped file containing the result of applying *sys.exp* to a nested slotfiller or nested association list that is a valid right argument of *t.open*. That right argument is then substituted for the Null in this call to *t.open*. Otherwise, the table consists of all files in that directory, except files named after any of the table variables associated with static grouping and static links.

- An association list right argument has symbol, value pairs in which the symbol is the name of a column in the table and the value is a character vector with one of the following forms:
  - `'n⌶filename'`
    meaning that the column is a mapped file in the usual sense if *n* is 0, 1, or 2, or
    - if *n* is -1, the column is *sys.imp* applied to the mapped file;
    - if *n* is -2, the column is ⍎ applied to the mapped file;
    - if *n* is -3, the column is ⊢ applied to the mapped file, i.e., the column is an unmapped copy of the mapped file.
  - `'n⌶filename0,filename1, ... ,filenamek'`
    meaning that the column is formed by using the rules described for `'n⌶filename0'` in the previous case, then the expression `column[,]←n⌶filename1` is evaluated, using the same rules for `'n⌶filename1'`, and so on, until all files through *filenamek* are appended to the column in this way.
  - `':expression'`
    meaning that the column is a dependency defined by *expression*.
  - `'[i]:expression'`
    meaning that the column is an itemwise dependency, with index *i*, defined by *expression*.
  - `'⍎expression'`
    meaning that the column is a variable whose value is the value of *expression*.
- The values of a slotfiller right argument have the same meanings as those of an association list.
- A nested association list right argument has symbol, value pairs in which the symbol is either the name of a column in the table or `` `. ``, in which case the value refers to the entire column, and the value is also an association list, whose symbol, value pairs are as follows:
  - if the symbol is `` `. `` then the value is a character vector specifying the value of the column, in one of the ways described in the previous case.
  - if the symbol is *`t.calendar* then the value is a vector of dates for a time series field (see *t.calendar*). When specified in *t.open*, it can also be a character vector of the form 'begin date → end date', e.g. `'1/1/89 → 12/31/89'`, from which the vector of dates will automatically be computed using *t.calendar*.
  - *if the symbol is `t.close* then the value is a character string holding an A+ expression that is executed after the column (or table) is closed.
  - if the symbol is *`na* then the value is the NA value to be used for this column.
  - if the symbol is *`t.open* then the value is a character string holding an A+ expression that is executed when the column (or table) is opened.
  - if the symbol is *`t.selector* then the value is a selector for this column (see "Selectors").
  - if the symbol is *`t.visible* and the value is 0 then this column is not included in _T_ when *t.table* is executed, while if the value is 1 it will be (even if it is not a valid screen table column).
  - if the symbol is *`t.sent* then the value is a symbol holding the name of the table from which this column is to be sent.
  - any attribute from the list *s.attributes `tableField*.
  - any other symbol represents an application-defined attribute and the value associated with the symbol becomes the value of the attribute.
- The values of a nested slotfiller right argument have the same meanings as those of a nested association list.

Note that once an attribute is given a value in *t.open*, that value can be retrieved with *_get*. Attributes can also be given values directly with *_set*, as well as with *t.open*.

The result of the function is the name of the opened table.

Examples

1. From the root context, open the table *x* whose columns are the existing variables *x.y* and *x.z*:

   ```
   `x t.open `y `z
   ```

2. From the *x* context, open the table *x* whose columns are all existing variables (in the *x* context):

   ```
   `x t.open ()
   ```

3. Open the table *x* whose column *y* is to be defined as ⍳100:

   ```
   `x t.open (`y;'⍳100')
   ```

   Note that when a column with an unqualified name is defined, the variable is created in the context of the table (*x* in this case).

4. Open the table *x* whose column *y* is to be defined as ⍳100 and column *z* is to be defined by the dependency definition *w.z*2*:

   ```
   `x t.open (`y;'⍳100';`z;':w.z*2')
   ```

   or

   ```
   `x t.open (`y`z;('⍳100';':w.z*2'))
   ```

5. Open the table *x* whose column *y* is to be defined as ⍳100 with the NA value -1234:

   ```
   `x t.open (`y;(`;'⍳100';`na;¯1234)
   ```

   or

   ```
   `x t.open (`y;(` `na;('⍳100';¯1234)))
   ```

6. Open the table *x* whose column *y* is to be defined as ⍳100 with the application-defined attribute fill whose value is '*':

   ```
   `x t.open (`y;(`;'⍳100';`fill;'*'))
   ```

   or

   ```
   `x t.open (`y;(` `fill;('⍳100';'*')))
   ```

7. Open the table *x* whose column *z* is to be defined by the itemwise dependency definition [*i*]:*w.z*[*i*]*2:

   ```
   `x t.open (`z;'[i]:w.z[i]*2')
   ```

8. Open the table *x* whose column *z* is to be defined as the mapped file /usr/local/lib/data.m, opened for reading only:

   ```
   (`x;'/usr/local/lib/') t.open (`z;'0:data')
   ```

9. From the *x* context, open the table *x* whose column *z* is to be defined as the mapped file /usr/local/lib/data.m, opened for updating:

```
$cx x
() t.open (`z;'1I/usr/local/lib/data')
```

10. The current directory is the one displayed when `$pwd` is executed in the A+ session. Assuming that the current directory is `/usr/local/lib/` and the current context is `x`, open the table `x` whose column `z` is to be defined as the mapped file `/usr/local/lib/data.m`, opened for local updating:
```
'' t.open (`z;'2Idata')
```
11. Open the table `x` whose column `z` is to be defined as the `sys.imp` of the contents of the mapped file `/usr/local/lib/data.m`:
```
(`x;'/usr/local/lib/') t.open (`z;'¯1Idata')
```
12. Open the table `x` whose column `z` is to be defined as the ⍟ of the contents of the mapped file `/usr/local/lib/data.m`:
```
(`x;'/usr/local/lib/') t.open (`z;'¯2Idata')
```
13. Everything is the same as the previous example, except the column is also named `data`:
```
(`x;'/usr/local/lib/') t.open (`data;¯2)
```
14. Open the table `x` whose column `z` is to be defined as the mapped file `/usr/local/lib/data.m`, opened for reading only, to which is appended the mapped file `/usr/local/lib/data2.m`:
```
(`x;'/usr/local/lib/') t.open (`z;'0Idata,data2')
```

`t.load` is a synonym for `t.open`.

Define a Column `t.define{w;x}`

Arguments and Result

The left argument `w` is either a symbol scalar, a character vector, a two-element nested vector consisting of a symbol scalar and a character vector in either order, or the Null. The right argument `x` is a symbol vector, an association list, a nested association list, a slotfiller, or a nested slotfiller. The result is a symbol scalar.

Definition

This function is like `t.open` except that only one column can be created; if the table already exists, this column is appended to it, and otherwise the table is created with this column as its only column. The left argument `w` is the same as `t.open`. The right argument `x` is also the same as `t.open`, except that it specifies just one column. The result is a symbol scalar holding the fully qualified name of the defined column.

Example

See "Grouping by Intervals".

Detach a Column from its Sources `t.detach{w;x}`

Arguments and Result

The left argument `w` is a nonempty symbol vector. The right argument `x` is any array.

Definition

The left argument `w` holds a list of table names. The table named in the last element of `w` is where the execution of this function has effect, and is called the target table in this definition. Any table named as one of the other elements of `w` is called a source table in this definition. The target table was either created from the source table(s) by a selection function such as `t.only`, or is the partition side of a link to the source table. If `z` is any column in the target table that is defined by t as a dependency on a column in one of the source tables, then the effect of this function is to make `z` an ordinary variable with its value equal to the current value of the dependency.

The right argument is currently not used. The result equals the left argument when source tables are specified, and otherwise is the same as the result would have been had all source tables been specified.

This function is useful when the columns in source tables are large and no longer of direct interest once links and views have been formed. These columns can be detached from their targets, leaving the current values of the targets intact, and then expunged.

Direct Static Link `t.link_d{w;f}`

Arguments and Result

The left argument `w` is a two-element symbol vector and the right argument `f` is a symbol vector, a two-element nested vector whose items are symbol vectors of the same length, or the Null.

Definition

The definition is analogous to that of `t.link`, and the differences are discussed in "Static Grouping" and "Static Links".

Example

See "Static Grouping" and "Static Links".

Direct Static Summary `t.group_d{w;f}`

Arguments and Result

The left argument `w` is a two-element symbol vector, and the right argument is a symbol vector or scalar, or the Null.

Definition

The definition is analogous to that of `t.group`, and the differences are discussed in "Static Grouping".

Example

See "Static Grouping".

Disperse `t.disperse{w;s}`

Arguments and Result

The left argument `w` is a two-element symbol vector, and the right argument `s` is a symbol scalar or one-element symbol vector. The result is a symbol scalar or symbol vector.

Definition

This function is a utility function in t. It is t-like in its function, but not directly related to t-tables.

The first element of the left argument `w` is a source context, and the second element is a target context. The right argument `s` holds the name of a slotfiller or nested slotfiller in the source context, i.e., the slotfiller is `w[0]%s`.

In the case of a slotfiller, this function creates a dependency in the target context for every symbolic index `y`, whose name is the name held by `y`, and whose definition is `y⊃w[0]%s`. An assignment callback is defined on this dependency so that whenever the dependency is assigned a value, the slotfiller element `y⊃w[0]%s` is also assigned this value. In this way data organized in a slotfiller is dispersed to global variables, where it can serve as columns in t-tables or screen objects.

The result in the slotfiller case is `w[1]`.

In the nested slotfiller case, contexts whose names are prefixed by the name in `w[1]` are created for every level of the slotfiller, dependencies are created in these contexts, and assignment callbacks are defined, all analogous to the slotfiller case above. The result in this case is a list of names, as symbols, of the created contexts corresponding to the ordinary slotfillers at the bottom of the nested slotfiller.

Example

```
      x.b←10×x.a←⍳3
      x.slot←(`k`m;(x.a;x.b))
      `x `y t.disperse `slot

      _def `y.m
m:`m⊃x.slot
      y.m←-y.m
      `m⊃x.slot
 0 ¯10 ¯20
```

Dynamic Derived Table `t.always{w;e}`

Arguments and Result

The left argument `w` is either a a one- or two-element symbol vector, or a symbol scalar. The right argument `e` is a character vector or scalar, or the Null. The result is an integer scalar.

Definition

If the right argument `e` is a nonempty character vector or character scalar, this function has the same effect as `t.only`, except that the selection defined by `e` remains active. That is, if any column in the source table whose name appears in `e` changes, the selection is recalculated. If the right argument is the Null then this function has the same effect as `t.only`, and in addition any active connection is broken.

The result is the number of selected rows.

Dynamic Sorted Derived Table `t.sorted{w;x}`

Arguments and Result

The left argument `w` is either a one- or two-element symbol vector, or a symbol scalar, or the Null. The right argument is either a nonempty symbol vector or an association list. The result is an integer scalar.

Definition

This function is to `t.sort` as `t.always` is to `t.only`: a table is sorted just as it is by `t.sort`, but it is also sorted again whenever one of the columns named in `x` changes.

The result is the number of rows in the sorted table.

Find `t.in{a;b}`

Arguments and Result

The left argument `a` is a character matrix and the right argument `b` is a character vector or scalar. The result is a boolean vector of size `#a`.

Definition

This function is a utility function in t. The right argument holds a list of character strings, each of length `1#⍴a`, separated by commas. If a matrix `c` is formed with these character strings as rows, then `t.in{a;b}` equals `a∈c`.

This function can be used to form selections in the right argument of selection functions such as `t.only`.

Example

The selection `'dept_no∈3 3⍴"D11D21D31"'` in "Row and Column Selection" can be phrased in the more Englishlike `'dept_no in "D11,D21,D31"'`. See "Selectors" for more on Englishlike selections.

Fix a View `t.fix{d;b}`

Arguments and Result

The left argument *d* is a one-element symbol vector or a symbol scalar. The right argument *b* is a one-element integral vector or an integral scalar. The result is an integer scalar.

Definition

The table named by the left argument *d* should be a view in which row selections are done, e.g., with *t.also*. When *t.fix{d;1}* is executed the current view *d* is fixed; any further selections done in the view[2] will refer to no rows in the source of *d* other than those present at the time *t.fix{d;1}* was executed. A subsequent execution of *t.fix{d;0}* releases the fixed view, i.e., returns to the default setting.

Example

```
    src.a←⍳100
    `src t.only 'a<90'
90            ⍝ The number of rows in view
    `view t.fix 1
    `src `view t.only '20<a'
20
    `src `view t.also '80>a'
29            ⍝ Only 9 rows are appended, rows 81 through 89 of src
    `view t.fix 0
    `src `view t.also '80>a'
39            ⍝ Now rows 90 through 99 of src appear in view
```

Index *t.index{i;x}*

Arguments and Result

The left argument *i* is an integer vector and the right argument *x* is a symbol scalar holding the name of a t-table column. If *f* is the table column named by *x*, i.e., *f←%x*, then the result has shape *(⍴i),(1↓⍴f)* and type *∨f*.

Definition

This function is a utility function in t. It is used in the definitions of columns sent to views, which are defined formally as *i#x*, but are actually *t.index{i;x}*. It is like the system function *_index*, except that the NA value is not specified. Instead, the NA value is *t.na{x}*.

A+ applications can use *t.index* wherever *_index* would otherwise be used but defaulting to this definition of NA values makes sense.

Example

See "How t Manages Row and Column Selection".

Indirect Static Link *t.link_i{w;f}*

Arguments and Result

The left argument *w* is a two-element symbol vector and the right argument *f* is a symbol vector, a two-element nested vector whose items are symbol vectors of the same length, or the Null.

Definition

The definition is analogous to that of `t.link`, and the differences are discussed in "Static Grouping" and "Static Links".

Example

See "Static Grouping" and "Static Links".

Indirect Static Summary `t.group_i{w;f}`

Arguments and Result

The left argument `w` is a two-element symbol vector, and the right argument is a symbol vector or scalar, or the Null.

Definition

The definition is analogous to that of `t.group`, and the differences are discussed in "Static Grouping".

Example

See "Static Grouping".

Lightweight Column Definition `t.let{d;f}`

Arguments and Result

The left argument `d` is a symbol scalar or one-element vector, and the right argument `f` is a symbol scalar or nonempty symbol vector. The result is a symbol scalar or symbol vector with the same number of items as the right argument `f`.

Definition

The left argument `d` is the name of a table, and each symbol in the right argument `f` is the name of an existing global variable or dependency. These names can be qualified or unqualified, and if unqualified, the object is assumed to be in the table context of `d`. The effect of this function is to append the objects named in `f` to the table `d` as new columns; note that the definitions of these columns are independent of t, unlike columns specified in `t.open` or `t.define`. The result is a symbol scalar or vector holding the fully qualified name of every defined column.

Link and Send `t.relate{s;d;f;g;h}`

Arguments

The arguments `s` and `d` are symbol scalars. The argument `f` is a valid right argument of `t.link` when the left argument is `s,d`. The arguments `g` and `h` are valid right arguments for `t.send` when the left arguments are `s,d` and `d,s`, respectively.

Definition

`t.relate{s;d;f;g;h}` is equivalent to executing the following expressions:

```
(s,d) t.link f
(s,d) t.send g
```

```
(d,s) t.send h
```

Link Two Tables `t.link{w;f}`

Arguments

The left argument `w` is a two-element symbol vector and the right argument `f` is a symbol vector, or a two-element nested vector whose items are symbol vectors of the same length, or the Null.

Definition

The symbols in the left argument are the names of tables, either base or view. The symbols in a symbol vector right argument are the names of columns called link fields that appear in both tables named in the left argument. The effect of this function is to link the two tables on the specified link fields, which means that mappings are established between the two tables that govern the way columns are sent from one to the other.

If the right argument is a two-element nested vector, then the symbols in the first item identify the link fields in the table named in the first element of the left argument, and the symbols in the second element identify the corresponding link fields in the other table. A Null right argument breaks any existing links between the two tables named in the left argument.

See "Linking Tables" for details.

Example

See the examples that follow "Linking Tables".

Link, with Permutation `t.link_b{w;f}`

Arguments and Result

The left argument `w` is a two-element symbol vector and the right argument `f` is a symbol scalar, or a one-element symbol vector, or a two-element nested vector whose items are symbol scalars or a one-element symbol vectors.

Definition

This function, like `t.link`, establishes a link between the two tables named in the left argument. As with `t.link`, the first element of the left argument names the partition side of the link and the second element names the replication side, and the right argument names the link field(s). The differences in the two functions are that there can only be one link field here, and the mappings between the two tables are computed differently: this function should be used for tables wherever `b.p` should be used instead of dyadic `⍳`.

Let table `l` be the partition side and table `r` the replication side. Unlike `t.link`, this function requires that a sort vector for the link field(s) on the replication side be known, and that sort vector is the user-defined table variable `r._H_l`. Specifically, this vector must have the following meaning. Say the link field on the partition side is `p` and on the replication side is `r`. Then `r._H_l` must be a permutation vector so that the items of `r._H_l#r%p` are in nondecreasing order (see "Sorted Arguments to the b-Context Functions", for the definition of nondecreasing order). The t-defined table dependency `l._G_r` is defined by the expression:

```
b.pr{r_f;r._H_l;r%p}
```

The partition side mapping `l._J_r` is derived directly from `l._G_r` and `r._H_l`, and the replication side mapping `r._I_l` is then derived directly from `l._J_r`. See, for example, "Static Grouping".

NA Value `t.na{x}`

Arguments and Result

The argument `x` is a symbol scalar holding the name of a t-table column. The result is an array.

Definition

This function is a utility function in t. It is used to produce a substitute value for a column item when that column is indexed by t with an out of range value, which can occur when columns in tables are of different sizes. The substitute value is determined as follows:

- if the `` `na `` attribute of the column `x` has been set, use that value;
- otherwise, if the `` `t.sent `` attribute has been set and its value is the symbol `s` (see `t.open`) , and `s∪y` is the name of the source column of `x`, and if the A+ types of `x` and `s∪y` are identical , i.e., if `∨%x` equals `∨s%y`, use the NA value for `s∪y`, as determined by these same rules;
- otherwise, use `(∨%x)⊃t.NA`.

Populate a View Table `t.send{w;f}`

Arguments and Result

The left argument `w` is a symbol vector with at least two elements, or an array complying with the left argument specification of `t.open`. The right argument `f` is a symbol vector, or an association list, a nested slotfiller, or an array complying with the right argument specification of `t.open`. The result is a two-column symbol matrix.

Definition

This function is used to populate a view created by a selection function such as `t.only`, or `t.group`, or to send columns from one side of a link to the other. This is accomplished by specifying three things: the names of the columns in the source table(s), the names of the corresponding columns in the target table, and the definitions of the target table columns in terms of the source table columns.

The meaning of the left argument is:

- A two-element symbol vector left argument specifies the source table from which columns are sent (`0⊃w`) and the target table that receives columns (`1⊃w`).
- A symbol vector left argument with more than two elements is a special case related to `t.cat`, and the execution of `t.send` should have been preceded by an execution of `t.cat`. When `t.send` is used to populate the target of `t.cat`, not all the source tables of the target need be used, and the source tables can be given in any order. Only columns from the source tables appearing in this left argument are catenated, in the order in which they appear.

- Any other left argument, taken together with the right argument, must form a valid left argument, right argument pair for `t.open`, and the meaning of this function in that case is exactly the same as for `t.open`.

When the left argument is a symbol vector with at least two elements, the meaning of the right argument is:

- Every symbol in a symbol vector right argument is the name of a column to be sent from the source table(s) to the target table. A new column in the target table has the same name as its source column(s).
- A nested slotfiller right argument has symbol, value pairs in which the symbol is the name of a column in the target table, say `` `c ``, and the value is also a slotfiller, whose symbol, value pairs are as follows:
  - if the symbol is `` `_from ``, the value is the name of the column in the source table(s) to be sent to `` `c ``. In the default case, when this symbol, value pair is not present, the column(s) in the source table(s) are also named `` `c ``.
  - if the symbol is `` `_type ``, the value is either 0 or any nonzero scalar. If 0, the target column is defined to be a dependency according to the `` `_frame `` case below. If nonzero, there should be a single source column, it should be a dependency, and any columns in the source table on which it depends, if not present in the target table, should be sent to it. The target column is defined to be a dependency with exactly the same definition as that of the source column. The default case, when this symbol, value pair is not present, is the value 0 case.
  - if the symbol is `` `_frame ``, the value is a dependency frame for this column (see "Dependency Frames"), which is used as the basis for the dependency definition of this column in the target table. When this symbol, value pair is not present, the dependency definition is a default one, which is essentially:
    `t.index{_V;c}`
    (see "How t Manages Row and Column Selection").
  - if the symbol is `` `_func ``, this must be one of two cases:
    - the column is being sent to a view initialized by `t.group` or one of its variants;
    - the column is being sent to a partition side of a link.

    If so, the value part of the pair is a group function for this column (see "How Group Functions are Specified and How t Applies Them"). When this symbol, value pair is not present, the group function is "first item".

- The so-called *nested form* is a slotfiller right argument with symbol, value pairs in which the symbol is the name of a column received from the source table(s), say `` `c ``, and the value is a nested vector with at most four elements. Each element has the meaning of a value associated with one of the symbolic indices of the nested slot-filler case: `` `_from ``, `` `_type ``, `` `_frame ``, and `` `_func ``; which meaning is deduced from the type and contents of the element. If no element can be associated with a symbolic index, the default value associated with that symbolic index is used.

  Note that as a special case, when a value has only one element, that element does not have to be enclosed.

- An association list right argument has symbol, value pairs like those in either the nested slotfiller or nested form case.

A set callback is placed on each field that is sent. It can be removed by executing `fld _scb (;)`.

Examples

See the examples earlier in this chapter for various uses of `t.send`. Other cases when `t.send` is not equivalent to `t.open` are:

1. Send a column `csrc` from the source table `src` to the target table `ctar` and name it `csrc` in the target table:
   ```
   `src `tar t.send (`ctar;`csrc)
   ```
   or
   ```
   `src `tar t.send (`ctar;<`csrc)
   ```
   or
   ```
   `src `tar t.send (`ctar;(`_from;`csrc))
   ```
2. Adding to example 1, suppose that the target table was established by `t.group` and the group function `bf` is to be sent as well:
   ```
   `src `tar t.send (`ctar;(`csrc;bf))
   ```
   or
   ```
   `src `tar t.send (`ctar;(`_func`_from;(bf;`csrc)))
   ```

Random Sample `t.sample{w;n}`

Arguments and Result

The left argument `w` is either a symbol scalar or a one-element or two-element symbol vector. The right argument `n` is an integer or the Null. The result is also an integer.

Definition

If the left argument has two elements, the first element holds the name of an existing table, base or view, and the second element holds the name of a table to be created by this function. If the right argument is an integer, then the created table is a random sample of n rows of the existing table, and the result equals n. If the right argument is the Null, then the created table is a rowwise permutation of the existing table, and the result is the number of rows in either one.

A one-element left argument holds the name of an existing view. If the right argument is an integer, the effect is to restrict the view to a random sample of n rows, and the result equals n. If the right argument is the Null, the effect is permute the rows of the view, and the result equals the number of rows in the view.

Reset `t.reset{}`

Result

The result is a symbol vector.

Definition

Close all t-tables, i.e., execute `s t.close ()` for all existing t-tables `s`. The result is a list of names, as symbols, of the closed tables.

Restrict a Table `t.only{w;e}`

Arguments and Result

The left argument `w` is either a two-element symbol vector, a one-element symbol vector, or a symbol scalar. The right argument `e` is either a character or integral vector or scalar, or the Null. The result is an integer scalar.

Definition

The right argument `e` denotes a row selection in a source table that is realized in a target table. The source and target tables are specified in the left argument. The result is the number of rows selected.

The symbols in the left argument are the names of tables, either base or view. If the left argument has two elements, then the first one denotes the source table of the selection and the second one the target table; if it has one element, it denotes the target table. When the left argument has two elements and the target table does not exist, it is created; when the argument has one element, it is assumed to be an existing table.

The target table cannot be a base table, i.e., it cannot be one created by `t.open`. A view table can serve as either the source and target table.

In the case of a two-element left argument, the right argument defines a selection of the rows in the source table that will make up the target table. If the right argument is an integer vector or scalar it is assumed to hold the indices of those rows in the source table. If the right argument is a character vector or scalar it is assumed to hold an A+ integer-valued expression; calling the value of this expression `v`, the target table will consist of the source table rows with row indices `v/⍳#v`.

In the case of a one-element left argument, the right argument defines a selection of rows in the existing target table. If the right argument is an integer vector or scalar, it is assumed to hold the indices of rows in the source table that are currently in the target table, while if the right argument is a character vector or scalar it defines a selection of rows from the current target table.

When the right argument is the Null, the resulting view contains all rows of the source, no matter whether the left argument has one or two elements.

When the right argument is an integer vector or scalar it is most likely the result of an A+ expression, and when it is a character vector or scalar it contains an A+ expression. The difference between the two forms of right argument is due to the way t evaluates a character right argument. For example, suppose a selection is to choose all rows for which the value of the column named col is greater than 10. Then a character right argument should be `'col>10'` or an equivalent statement, and should not refer directly to the source table. See the utility function *t.in* for another example.

Note that `t.only` simply defines view tables; it does not populate them with columns. For that purpose, see *t.send*.

Example

See "Row and Column Selection". Note that:

```
`employees `p_employees t.only '18⍴1 0'
```

has the effect of creating the view *p_employees* with rows from the *employees* table with indices `(18⍴1 0)/⍳18`. The equivalent selection can be made with the latter expression as the numeric right argument to *t.only*, as in:

```
`employees `p_employees t.only (18⍴1 0)/⍳18
```

Screen Table *t.table{w}*

Arguments and Result

The argument w is either a one- or two-element symbol vector. The result is a symbol scalar.

Definition

If the argument *w* has one element then it is either a qualified or unqualified name; if unqualified, it is the name of a t-table, whereas if qualified, it is the name of a variable in the context of a t-table. An unqualified name is treated as if it were the qualified name *w∪_T_*. A two-element argument *w* is treated as if it were the qualified name *w[0]∪w[1]*. The result is this qualified name. An object with this name is created that contains the same value as that described for *_T_* in the table "t-Created Variables, Dependencies Common to All Table Contexts". And, finally, this object is bound to the table display class (in particular, this function assumes that the s context has been loaded).

Example

See "Successive Multiple Group Fields".

Sort *t.sort{w;x}*

Arguments and Result

The left argument *w* is either a one- or two-element symbol vector, or a symbol scalar, or the Null. The right argument is either a nonempty symbol vector or association list or the Null. The result is an integer scalar.

Definition

The symbols in the left argument are the names of tables, either base or view. If the left argument has two elements then the first one denotes the source table of the selection and the second one the target table; if it has one element, it denotes the target table. When the left argument has two elements and the target table does not exist, it is created; when it has one element, it is assumed to be an existing table.

The target table cannot be a base table, i.e., it cannot be one created by *t.open*. A view table can serve as either the source and target table.

If the left argument *w* has one element and the right argument *x* is a nonempty symbol vector or symbol scalar, the effect of this function is to sort the table named by *w*, as follows. The A+ primitive Grade Up is applied to the column named by the last element in *x*, and the resulting permutation vector is applied to all columns of the table. Grade Up is then applied to the column named by the next to last element in *x*, and the resulting permutation vector is applied to all columns of the table. This process continues until Grade Up has been applied to all columns named in *x*, in their right to left order of appearance.

If instead the right argument *x* is an association list, then the symbol in each symbol, value pair is the name of a column, and the value is either 0 or 1. The sorting process described above is modified so that Grade Up is applied to a column only when its associated value is 1, and otherwise Grade Down is applied.

When the left argument has two elements, the first element names a source table and the second element a view of that source. The effect of this function is apply the sorting process to the source table to obtain a permutation vector, although that table is left unchanged; the permutation vector is applied to the rows of the view table instead.

If the right argument is the Null then a table is returned to its unsorted order.

Successive Grouping *t.report{w;f}*

Arguments and Result

The left argument *w* is a two-element symbol vector. The right argument *f* is a symbol scalar or nonempty symbol vector.

Definition

The symbols in the left argument hold the names of tables. As in *t.group*, the first element names an existing base or view table, called the source table, and the second element names a view table to be created by this function, called the target table. The symbols in the right argument hold the names of columns in the source table. Unlike *t.group*, where grouping is done on the rows of the collection of columns named in the right argument, here the grouping is done successively on the individual columns, starting with the first one named in right argument. That is, all rows with common values in the first column named in the right argument are grouped, just as in *t.group* with only one column named in the right argument. Then within each such group, all rows with common values in the second column named in the right argument are grouped, and so on. The effect of this function is to form a series of intermediate tables between the source and target tables, one for each column named in the right argument.

As with other t functions, the view created by *t.report* must be populated with columns from the source table, using *t.send*. In this case, however, t contributes a column of row headings indicating the successive grouping. This column is named _ and is the first name in *_T*.

Example

See "Successive Multiple Group Fields".

Summarize a Table *t.group{w;f}* or *t.break{w;f}*

Arguments and Result

The left argument `w` is a two-element symbol vector, and the right argument is a symbol vector or scalar, or the Null, or a slotfiller.

Definition

The left argument holds the names of the source table and destination table.

The symbols in the left argument are the names of tables. The first element names an existing base or view table, called the source table, and the second element names a view table to be created by this function, called the target table. The symbols in the right argument, including the case of the symbolic indices in a slotfiller, are the names of columns called group fields that appear in the source table; all fields are group fields if the right argument is the Null. The effect of this function is to establish a onesided link from the source table to the target table on the specified group fields, with the source table acting as the partition side of the link, which means that mappings are established between the two tables that govern the way columns are sent from one to the other. See "Group Fields and Group Functions" for details. In the case of a slotfiller right argument, every value `` `a⊃f `` is an integer vector specifying the group-by-interval values for the field `a`.

`t.break` is a synonym for `t.group`.

Summarize and Send `t.partition{s;d;f;g}`

Arguments

The arguments `s` and `d` are symbol scalars. The argument `f` is a valid right argument of `t.group` when the left argument is `s,d`. The argument `g` is a valid right argument for `t.send` when the left argument is `s,d` .

Definition

`t.partition{s;d;f;g}` is equivalent to executing the following expressions:

```
(s,d) t.group f
(s,d) t.send g
```

Union `t.also{w;e}`

Arguments and Result

The left argument `w` is either a one- or two-element symbol vector or a symbol scalar. The right argument `e` is either a character or integral vector or scalar, or the Null. The result is an integer scalar.

Definition

The right argument `e` denotes a row selection in a source table that is appended to a target table. The source and target tables are specified in the left argument. The result is the number of rows selected.

The symbols in the left argument are the names of tables, either base or view. If the left argument has two elements, then the first one denotes the source table of the selection and the second one the target table; if it has one element, it denotes the target table. When the left

argument has two elements and the target table does not exist, it is created; when the argument has one element, it is assumed to be an existing table.

The target table cannot be a base table, i.e., it cannot be one created by `t.open`. A view table can serve as either the source and target table.

In the case of a two-element left argument, the right argument defines a selection of the rows in the source table that will be appended to the target table. If the right argument is an integer vector or scalar, it is assumed to hold the indices of those rows in the source table. If the right argument is a character vector or scalar, it is assumed to hold an A+ integer-valued expression; calling the value of this expression `v`, the source table rows to be appended to the target table are those in `v/⍳#v` not already in the target table.

In the case of a one-element left argument, the right argument defines a selection of rows in the complement of the current target table, i.e., rows in the source table that are not in the target table. If the right argument is an integer vector or scalar, it is assumed to hold the indices of rows in the source table that are currently in the complement of target table, whereas if the right argument is a character vector or scalar, it defines a selection of rows in the complement of the current target table. The rows selected from the complement are appended to the current target table.

In the case of a two-element left argument, any unqualified table column names in the right argument refer to the source table, whereas for a one-element left argument, they refer to the target table. In the latter case, the columns so named must have already been sent to the target table.

When the right argument is the Null, the resulting view contains all rows of the source, no matter whether the left argument has one or two elements.

When the right argument is an integer vector or scalar, it is most likely the result of an A+ expression, and when it is a character vector or scalar it contains an A+ expression. The difference between the two forms of right arguments are due to the way t evaluates a character right argument. For example, suppose a selection is to choose all rows for which the value of the column named col is greater than 10. Then a character right argument should be `'col>10'` or an equivalent statement, and should not refer directly to the source table. See *t.in* for another example.

Note that `t.also` simply defines view tables; it does not populate them with columns. For that purpose, see *t.send*.

Example

See "Row and Column Selection".

View and Send `t.view{s;d;f}`

Arguments

The arguments `s` and `d` are symbol scalars. The argument `f` is a valid right argument of `t.send` for the left argument `s,d`.

Definition

`t.view{s;d;f}` is equivalent to executing the following expressions:

```
(s,d) t.only ()
(s,d) t.send f
```

Table Variables and Dependencies

t-Context Global Objects

Name	Description
`t.CONNECT`	This function is either ∨ or ∧; see "Selectors" for its use.
`t.NA`	This variable is a slotfiller holding the default NA values. It is: `(`int `float `char `sym `box `func `null;` `(¯999999999; ¯999999999.; ' '; `; <(); <{⊢}; <())`
`t.TABLES`	This variable holds a list of names, as symbols, of all the t-tables created in this A+ session.

The variables described in the following table appear in every table context created by t. They are referred to by fully qualified names only when necessary.

t-Created Variables, Dependencies Common to All Table Contexts

Name	Description
`_A`	This variable holds the value `1⍴<V` in the specification of the following (see "Grouping by Intervals"): `` `src `this_table t.group (`group_field;V) `` In general, this variable is a nested vector of integer vectors whose items are grouping by intervals specifications corresponding to the columns in `_T`.
`_D`	This variable holds a list of names, as symbols, of the target tables for which this table is a source.
`_D_`	This dependency is a nested slotfiller of the items in `_D`; the top level is the name of this table; the next level holds the names of the views created directly from this table; the next level holds the names of the views created directly from those views; and so on. Display this dependency as a tree object for a convenient representation.
`_F`	When `t.fix` is executed with the right argument `1` for this table, this variable is assigned the current value of `_V`. Initially, and when `t.fix` is executed with the right argument `0` for this table, this variable is set to the Null.
`_G_r`	This dependency is defined by t when this table is on the partition side of a link made with `t.link_b`. Let `l` denote this table and let `r` denote the table on the replication of the link. This dependency is computed from `b.pr` and the sort vector `r._H_l`, and is the first step in the efficient computation of the mappings between the two sides on the link. See `t.link_b`.

_H_l	This dependency is user defined for t.link_b, t.link_d, and t.group_d, but defined by t for t.link_i and t.group_i. Its value is a permutation vector used in the definition of the map from the source (this table) to the view in the case of grouping, and in the case of links, from the replication side (this table) to the partition side. The view or partition side is table l.
_I_l	This dependency defines the mapping from the partition side of a link (the table l) to the replication side (this table). See "How t Manages Table Links". Note that the name of this mapping is specific to these two tables, so any table can be linked to several others at the same time.
_J	When this table is created by t.group, this dependency defines the mapping from the source table to this table. See "How t Manages Group Fields".
_J_r	This dependency defines the mapping from the replication side of a link (the table r) to the partition side (this table). See "How t Manages Table Links". Note that the name of this mapping is specific to these two tables, so any table can be linked to several others at the same time.
_K	When this table is created by t.group, the value of this dependency is the first element of every item of _J. This dependency is used to implement the default group function for a column sent to this table.
_K_r	This value of this dependency is the first element of every item of _J_r. When this table is serving as the partition side of a link, this dependency is used to implement the default group function for a column sent to this table.
_L	The value of this dependency is a list of names, as symbols, of those tables on the partition side of a link with this table.
_L_	The value of this dependency is a two-column symbol matrix whose rows hold the names, as symbols, of partition side, replication side tables in table links. All links with this table as the partition side appear in this matrix; for all tables on the replication side of a link with this table, all links with those tables as the partition side also appear in this matrix; and so on (until the transitive closure is complete).
_M	The value of this dependency is the depth (monadic ≡) of the column named by 0#_T; it is the depth of this table if all columns have the same depth.
_N	The value of this dependency is the number of items in the column named by 0#_T; it is the number of rows in this table if all columns have the same number of items.
_O	This variable is a two-element nested vector that is set for this table if it is created with t.open; the first element holds the directory specified in the left argument of t.open (or the default), and the second element holds the field specifications in the right argument.
_R	The value of this dependency is a list of names, as symbols, of those tables on the replication side of a link with this table.
_R_	The value of this dependency is a two-column symbol matrix whose rows hold the

	names, as symbols, of replication side, partition side tables in table links. All links with this table as the replication side appear in this matrix; for all tables on the partition side of a link with this table, all links with those tables as the replication side also appear in this matrix; and so on (until the transitive closure is complete).
`_S`	This variable holds a list of names, as symbols, of the source tables for which this table is a target; see `t.group`, `t.cat`, and `t.only`. If `_S` is identical to the Null, then this table is a base table. (More than one source table can occur with table catenation.)
`_S_`	The value of this dependency is a list of names, as symbols, of the base tables that are the ultimate sources of this table, i.e. from which the creation of this table can be traced through a series of applications of `t.group`, `t.cat`, and `t.only`.
`_T`	This variable holds a list of names, as symbols, of the column variables in this table.
`_T_`	`t.table` should be executed to set this variable, in which case if `_U` is the Null, this variable is set to a list of names, as symbols, of the column variables in this table that conform to the screen management definition of tableField. If `_U` is not the Null, then `t.table` sets this variable to equal `_U`.
`_U`	This variable holds a list of names, as symbols, of the column variables in this table that are to appear on the screen, as determined by the application. It is set by the user.
`_V`	This variable is the mapping from a source table to this view table. See "How t Manages Row and Column Selection".
`_V_`	When this view table has one source table, the value of this dependency represents the complementary view, in that every row of the source table is selected by `_V` or `_V_`.

12. Calling C Subroutines from A+

This chapter describes how to write C programs to be called from within A+, and what you have to do within A+ to call those functions.

How to Compile C Functions to Be Called by A+

Writing functions to be called from A+ is straightforward. You must:

1. Write the C function normally, with `#includes` for the `.h` files described below. Several functions may be included in the file. The file should *not* have a `main()` function in it!

   It is a good idea to make all functions "static" (by putting the keyword `static` before the function name) unless they are directly called from A+. That is, all subroutines should be made static. This prevents their names from cluttering up the A+ namespace, which might cause a problem if another dynamic load contains a subroutine with the same name. All nonlocal variables *must* be static.

2. Compile the C source code into an object file (*not* an executable file). This is done with the `-c` argument to `cc`. E.g.,

```
cc -c -o foo.o foo.c
```

compiles the C source file `foo.c` into `foo.o`.

You may compile your code into several object files. If you do this, remember that static functions can be called only from another function in the same object file. But see "Another Way To Call C Routines From A+: Static Link".

How to Use C Functions When You Are in A+; Dynamic Loading

Once the functions are written, you must either dynamically load or statically link them into A+ before you can call them. In order to dynamically load them, you must know the names and paths of the object files, and the names of the C functions you want to call (known as the "entry points"). An object file can have several entry points, but you need to know only the names of the entry points you are loading. Static functions cannot be entry points.

Warning! If the A+ name used in `_dyld` (in its right argument) begins with an underscore, then the function will be installed in the root context, no matter what the current context, and it will be listed by `$sfs` but not by `$xfs`.

The procedures you must follow depend upon the system you are going to use.

Dynamic Loading on Sun Machines

When a C program gets compiled, the compiler puts an underscore (_) in front of the name of each function. So if you write a routine called `look()`, the entry point will be `'_look'`.

Programs are dynamically loaded using the system function `_dyld`, which takes two arguments. The first (or left) is the name(s) of the object file(s) as a character string. Multiple object files are separated by spaces. All of the subroutines referenced by the entry points must be in one of the object files, or part of A+. You cannot refer to subroutines or entry points already dynamically loaded.

The second (right) argument is a nested array in the form

```
(entry; name; args)
```

or, to load more than one function from the same file(s),

```
(entry1; name1; args1;
 entry2; name2; args2;
 ...                  ;
 entryN; nameN; argsN)
```

where `entryI` is the entry point name, as a character string, `nameI` is what you want to call the function in your workspace (the A+ name, also as a character string), and `argsI` is a numeric vector describing the types of the arguments. (This vector is described below.)

For example:

```
    'test.o' _dyld ('_look'; 'lookat'; 9 0)
    $xfs
lookat
```

Dynamic Loading under AIX

The mechanism provided by IBM to support dynamic loading of object code into a running process dictates that three files be provided.

1. A file listing the symbols in the A+ interpreter which are to be accessed by the code to be loaded in. This file is provided for you and can be found in `/usr/local/bin/a+`*x.xx*`/lib/liba.exp` where *x.xx* is the A+ version and release, e.g., `/usr/local/lib/liba.exp`.

   For the current default release, you can use `/usr/local/lib/liba.exp`.

2. A file listing the symbols in the code to be loaded that the A+ interpreter needs to know about. You must create this file with an editor.
3. A single object file containing the modules that you wish to load in. This file is produced by the linker as described below and will be referred to as the "shareable object" file. By convention these files should have a `.so` suffix.

This is best illustrated by an example.

Below is the file `ref.c`, which returns the reference count of the A+ object passed into it.

```
/* begin ref.c */
#include <a/arthur.h>
I ref(a)   A a; {
   return a->c;
}
/* end ref.c */
```

Let's say you want to dynamically load in this function and also the function `dswap()` from the BLAS library. The following procedure should be followed.

1. Create the `exports` file, which enumerates all symbols that you want A+ to know about. The first line of this file contains the full pathname of the `.so` (shareable object) file which will be dynamically loaded. Subsequent lines enumerate symbols to be exported to A+, one per line. In this case we can create the file named `xmpl-exports`:

```
2.       #!/u/foobar/src/hodedo/xmpl.so
3.       ref
        dswap
```

4. Compile all of your C and FORTRAN sources. In this case we only need to compile `ref.c` to produce `ref.o`.

```
cc -c ref.c
```

5. Link together all the object (`.o`) files and libraries that contain modules which you want to load into A+. The link command must specify an entry point (with the `-e` option) for

the linked result because the default entry point is `crt0`, which is already the A+ interpreter's entry point. For the entry point, use any function name you wish that is in the code to be loaded. The link command must also specify the two files describing the symbols to be imported or exported. In this case we need to link `ref.o` and `libblas.a`:

```
cc -e ref -bI:/usr/local/lib/liba.exp -bE:xmpl-exports
    ref.o -o xmpl.so -lblas
```

6. Dynamically load the code into the A+ session. This should occur almost instantaneously.

```
7.      "xmpl.so" _dyld ("ref";"ref";0 9;
                         "dswap";"dswap";V_,4,FP,4,FP,4)
```

Notes:

- Do not dynamically load the same `.so` more than once into the same A+ session. This can cause A+ to crash - and sometimes the machine.
- The left argument to `_dyld` must be a single `.so` file name.
- You do not need to put a leading '_' on C symbol names.
- If you are running a version of the A+ interpreter other than `/usr/local/bin/a` then you will need to use an export file other than `/usr/local/lib/a-exports` because the first line of the file has to have the path hardcoded. The file `/usr/local/lib/aX-exports` has been provided for aX users.

Another Way To Call C Routines From A+: Static Link

A hook that was added allows you to create a new A+ executable file with your C or C++ code linked in. The main use for this is in debugging code that will later be dynamically loaded.

The hook is an empty function called `uextInstall()`. It is invoked in `/u/aplus/3prod/src/main/aplus_uext.c`. To use this facility to load your functions into A+, you need to add `install()` calls to `uextInstall()`.

Then you compile and link, to create an A+ executable file. If you compile with the debug flag, you can run the new A+ executable file under a debugger and have access to your own code.

The Basic A+ Data Types

The include file `a/arthur.h` defines the basic data types which A+ employs. These are: `I` - long; `F` - double; `C` - char. When dealing with A+ objects, these typedefs should be used to refer to integers, floating-point numbers, and characters, respectively.

An A+ object (a variable in an A+ "workspace") has the following typedef:

```
typedef struct a{ I c, t, r, n, d[MAXR], i, p[1];};
```

`c` **- reference count.**

How many pointers to this object exist? This helps determine whether an object can be modified in place, or whether a copy of the object must be created. If this number is 0, the object is a mapped file, and cannot be written to directly. ("Mapped Files" tells how to write to mapped files.)

An A+ object should be modified only if `c` is 1.

`t` **- type.**

What are the elements of the object? They should be one of the following values, which are `#defined` in `a/arthur.h`: `It`, `Ft`, `Ct`, `Et`, or `Xt`. `It`, `Ft`, and `Ct` refer to integers, floating-point numbers, and characters (`I`, `F`, `C`). Nested arrays and symbols are type `Et`. `Xt` is used for "executable types" - functions and operators. These are beyond the scope of this chapter.

`r` **- rank.**

The number of dimensions of the object.

`n` **- number of elements.**

The number of elements in the data array (`p`). With the type (`t`), it determines the size of the A+ object.

`d[]` **- dimensions.**

An array of the dimensions (`⍴`...) of the object. *`MAXR`* is the largest rank allowed (currently 9).

`i` **- items.**

The number of items in the object. It is the number reported by *`_items`*.

`p[]` **- data array.**

It is defined as having one element of type `I`, but that is just to fool the compiler. In fact, its actual length is determined by `n`, and the actual type is determined by `t`. It is worth noting here that, for objects of type `Ct`, `p[]` is always a null-terminated string, and has `n+1` elements. For all other types, `p[]` has `n` elements.

Since A+ objects are almost always allocated from dynamic memory, variables are more often than not pointers to A+ structures rather than the structures themselves. The type `A` is defined to be a pointer to an A+ object.

Nested arrays are A+ objects of type `Et`. For such objects, `p[]` is an array of pointers to the A+ objects which compose the array. Symbols are also represented as objects with type `Et`. For symbols, `p[]` is an array of pointers to another struct (the `s` struct). Symbols are somewhat more complicated than other A+ objects.

Reference Counts - a Closer Look

The reference count field is used to save memory and time by eliminating identical copies of variables. When a variable is assigned the value of another variable, that variable is normally not

copied. Instead, the new variable name is set to point at the same A+ object, and the reference count of that object is incremented.

Objects with reference counts greater than one are pointed to by more than one variable and should not be changed. You must duplicate the object instead (and decrement the reference count for the original object).

When you "dereference" an object - by expunging a variable, or dropping elements from a linked list - the reference count is decremented. If it becomes zero when decremented, it is destroyed, and the associated memory is freed.

The function `ic(aobj)` is used to increment a reference count, and `dc(aobj)` is used to decrement it (and possibly erase the object). They work recursively on nested objects.

`dc()` is rarely used in C subroutines. `ic()` is used primarily when modifying or returning arguments passed to the function. (See below.)

The Argument Vector

In A+, all functions must have a fixed number of arguments (a number not exceeding 9). This is also true for C functions called by A+. (This fixed number for a C function to be called from A+ cannot exceed 8.) In addition, C functions often expect only certain kinds of arguments - integers, for example - and behave badly if they receive an argument they do not expect.

The argument vector describes the number and types of the arguments to the C function, and the result which it returns. Also, A+ provides several different ways to pass data from A+ to C, which simplifies the C programs you must write. The argument vector allows you to select among these ways.

Theory

The argument vector is composed of numbers between 0 and 15. The first number describes the result of the function, if any. If there is no result, use code 8, as described below. Otherwise use codes 0, 7, or 9.

The remaining numbers describe the arguments to the function. The length of the vector determines how many arguments there are. The maximum number of arguments allowed is eight.

The sixteen codes are shown in the table "C-Function Argument Types".

C-Function Argument Types

Code	Meaning (an asterisk means acceptable for a result)
0	any A+ object *
1	A+ object consisting of integers
2	A+ object consisting of floating-point numbers
3	A+ object consisting of characters

4	data array of any A+ object
5	data array of A+ object consisting of integers
6	data array of A+ object consisting of floating-point numbers
7	data array of A+ object consisting of characters *
8	First element of data array (use only for *void* result) *
9	single integer *
10	single floating-point number (*don't use*)
11	single character (*don't use*)
12	unique copy of any A+ object
13	unique copy of A+ object consisting of integers
14	unique copy of A+ object consisting of floating-point numbers
15	unique copy of A+ object consisting of characters

* The result must be one of the codes marked with an asterisk.

Codes 0-3 pass a pointer to the A+ structure. Codes 4-7 pass a pointer to the data array within the A structure (`aobj->p`). Codes 8-11 pass the value of the first element of the data array of the A+ object (`*aobj->p`). This does not work correctly for characters and floating-point numbers, which have a different size. Codes 12-15 pass an A+ object whose reference count is guaranteed to be 1. This means that you can modify the object without causing adverse side effects.

Practice

Although all 16 codes are defined, not all of them are useful. In fact, if you need information about the rank and type of your argument, only types 0 and 12 should be used.

If you need to know anything about the shape of the argument, the entire A+ object must be passed. This limits you to types 0-3, or 12-15, which are used only in special circumstances (see below). Using types 1-3 causes the interface to return a type error if you are passed the wrong kind of data. It also coerces floating-point numbers to integer (provided that `1|`*data* is 0). You must do any checking for rank or length yourself.

Since argument types 4-7 and 8-11 do not pass you the entire A+ object, you cannot check rank or the number of elements of the object.

Argument types 4-6 are problematic. They pass you a pointer to the data array, but you have no way of knowing the size or dimensions of the array. Type 7 (character) is useful, and will pass you a character string, which is null-terminated. You will, however, lose all shape information, so a vector of length 15 will appear identical to a 3 by 5 matrix.

Argument types 8-11 are designed for single-element arrays. (Anything else generates a rank or length error.) This is currently defined only for integers, so types 10 and 11 should not be used. When passing or returning a scalar integer, use type 9. When a function does not return a result, use type 8.

Types 12-15 are used when you want to make an internal modification to the A+ structure passed, and then return the result. An argument passed this way can be safely modified.

Returning a Result from a C Function

Most C functions called by A+ return a result. The argument type for the result must be 0 (an arbitrary A+ object), 7 (a character string), or 9 (an integer scalar). If your function does not return a result, it should be declared as "void", and the return type should be 8.

To return a scalar integer (type 9), just use the return command, e.g., `return(7)`.

To return a character string, also use the `return` command. Be sure to declare your function as returning a `char*`. E.g., `char *hw() { return("Hello, world!"); }`.

Note that A+ takes a copy of the string you return, so you are responsible for freeing any strings you create with `malloc()`, `ma()`, or `strdup()`. In general, you can free the string just before returning it, and this will work fine. `ma()` performs atmp memory allocation, where the argument specifies the number of words, and returns `I*`.

If you are returning an argument as the result, you must use `ic()`. See "Modifying and Returning Arguments".

If you are returning an A+ object (arg type 0 and not an argument to the function), you must create the appropriate object. The next section describes how to do this. Declare your function as returning an "A" type - a pointer to an A+ object. For example,

```
A foo(x,y)
```

If you are returning an A+ object, `return(0)` causes a null to be returned. Returning 0 can also be used to indicate an error condition.

Creating A+ Objects

A+ provides several functions to create A+ objects. You must know the size and type of an A+ object before you create it. There are several functions to make it easier to create common A+ objects, such as vectors, or integer scalars.

Initialized object

```
A gi(i) I i;            /* make a scalar integer */
A gf(f) F f;            /* make a scalar float */
A gsv(x,s) I x; C *s;   /* make a string; x is 0 (raw), 1 (apl), or 2 (c) */
A gc(t,r,n,d,p)I t,r,n,*d,*p;   /* make an A+ object,
                                   copying data from p */
```

Uninitialized object

Creating A+ Objects

Expression	Effect
`A gs(t) I t;`	make a scalar
`A gv(t,n) I t, n;`	make a vector
`A gm(t,d1,d2) I t,d1,d2;`	make a matrix (2-dimensional array)
`A ga(t,r,n,d) I t,r,n,*d;`	make an array (`r` dimensions)
`A gd(t,a) I t; A a;`	make an object taking `r`,`n`,`d` from `a`. `gd(t,a)` <=> `ga(t,a->r,a->n,a->d)`

The argument names, in all cases, conform to the A+ structure described above. `t` is type, `r` is rank, `n` is the number of elements, `d` is the array of dimensions. New A+ objects always have reference counts of one.

Because creating A+ objects involves memory allocation, whenever you create an A+ object you must later either destroy it with `dc()` (see below), or return it, either alone or as part of a nested array.

Memory Allocation - What to Do and What Not to Do

A+ includes its own memory allocation functions for atmp: `ma()`, `mab()`, and `mf()`. They work pretty much like `malloc()` and `free()`, *except* that `ma()` takes a number of words as an argument rather than a number of bytes (1 word = 4 bytes) and that `ma()` and `mab()` use atmp and `malloc()` uses the heap, and therefore should probably be limited to small allocations, under 1K, say.

For portability, use `mab()` or `malloc()`, and cover your allocation and deallocation routines, checking for errors such as no more space. If you do use `ma()`, be careful! Remember it takes an argument in words, not bytes.

Anything you allocate with `ma()` or `mab()` you must free with `mf()`. Anything that you allocate with `malloc()` or `strdup()` must be freed with `free()`. Don't mix them up. (This is another good reason to stick with `mab()`.)

To "erase" an A+ object, call `dc(aobj)`, *not* `mf(aobj)`. This should be rare, since you should not erase arguments to your function, so the only A+ objects you erase should be ones that you created earlier in the function. This shouldn't come up too often.

Modifying and Returning Arguments

In general, C routines called from A+ are expected to behave like A+ routines - all arguments are call by value. This means that the arguments should not be modified, since that would cause unexpected side effects in the A+ workspace.

However, if you use argument types 12-15, you can safely modify the arguments to the function. These types guarantee that the argument has a reference count of 1.

When you create your own A+ object, using `ga()` for example, you can simply return the created object when your program exits. This is not true for modified arguments, or arguments returned as part of a nested array. To return a modified argument, or incorporate an argument as part of a nested array, you must run `ic()` on the object.

The reason is that the function `dc()` is run on all arguments after your program exits. This function causes the arguments to be erased unless you increase the reference count with `ic()`. If you forget to do this, values of variables in the A+ workspace will be changed randomly.

Since `ic()` is defined as returning an integer, you will often want to cast the result to type A. If you don't do this, you will get the compiler warning "illegal combination of pointer and integer".

Examples of Modifying and Returning Arguments

Example 1: `join`

Let's say we want to write a function, `join()`, that takes two A+ objects and returns a nested array containing the two elements. That is, *`join{a;b}`* is the same as `(a;b)`.

In C, we would write:

```
A join(obj1, obj2)
  A obj1, obj2;
  {
   A result=gv(Et, 2);       /* create nested vector of length 2 */
   result->p[0]=ic(obj1);    /* load result vector with objs,
                                incrementing reference count */
   result->p[1]=ic(obj2);
return(result);              /* return result, not incremented
                                because we created it in this function */
}
```

After compiling the function into `join.o`, we would then enter in A+:

```
     'join.o' _dyld ('_join';'join';0 0 0)
```

We can now use *`join`* as a function in the workspace:

```
     7 join 'abc'
<  7
< abc
```

Example 2: `clone`

Now we want to write a function that takes an arbitrary A+ object (*`aobj`*), and an integer (*`n`*), and returns a nested array containing *`n`* copies of *`aobj`*. That is, *`clone(aobj, n)`* is equivalent to *`n ⍴ <aobj`*. Notice that we increment the reference count each time we insert *`aobj`* into the nested array.

In C we would write:

```
A clone(aobj, n)
  A aobj;
  I n;
{
    I i;
    A result=gv(Et, n);
    for(i=0;i<n;++i) result->p[i]=ic(aobj);
    return(result);
}
```

We would load in A+ by entering:

```
     'clone.o' _dyld ('_clone';'clone';0 0 9)
     'abc' clone 2
< abc
< abc
```

Example 3: `ravel`

Now let's write a function that modifies its argument. We will replicate the Ravel function (monadic comma). Whatever we get, we will turn into a vector. We could do this by copying the *aobj* into a vector:

```
A ravel1(aobj)
  A aobj;
{
    A result;
    /* make new a object */
    result=gc(aobj->t, 1, aobj->n, &aobj->n, aobj->p);
    return(result);
}
```

To load in A+:

```
     'ravel1.o' _dyld ('_ravel1';'ravel';0 0)
```

We can get a somewhat neater and faster function if we modify the argument in place. Thus:

```
A ravel2(aobj)
  A aobj;
{
    aobj->r = 1;          /* change argument in place */
    aobj->d[0]=aobj->n;
    return(ic(aobj));     /* increment rc of modified argument */
}

     'ravel2.o' _dyld ('_ravel2';'ravel';0 12)
```

Note that we must now use argument type 12, and increment the reference count on the result (because it will automatically be decremented upon this function's return). Note also that, since the argument that is being modified is of type 12, it may be either the argument that appears in the A+ expression calling the function or a copy of that object. It will be a copy unless the reference count is 1 for the object.

Signalling Errors

If you detect an infelicity in your function, you may want to cause the A+ process to suspend execution and indicate, for example, a length error. This is done using two external variables: `I q; C *qs;`

To report an error, set the value of `q` (and possibly `qs`), and `return(0)`. If your program returns 0, the A+ process will check the value of `q`. A nonzero value indicates an error condition.

Positive numbers represent different predefined error codes, as shown in the following table.

Error Codes

Code	Meaning	Code	Meaning
1	interrupt	10	index
2	wsfull	11	mismatch
3	stack	12	nonce
4	value	13	maxrank
5	valence	14	nonfunction
6	type	15	parse
7	rank	16	maxitems
8	length	17	invalid
9	domain	18	nondata (for an argument of any type other than 0, A+ will check for nondata; you must detect and handle wrongly nondata type 0 arguments)

If `q` is -1, the A+ process will report the error in the `qs` string.

The file `a/firca.h` contains `#defines` for these codes, as well as macros for reporting error conditions. For example:

```
if (a != b) ERROUT(ERR_LENGTH);       /* reports a length error */


if (positive(a)) ERRMSG("polarity");  /* reports a polarity error */
```

Note that these macros exit the function, so be sure to clean up first!

Executing A+ Expressions from Dynamically Loaded C Programs

Your C programs that have been dynamically loaded into A+ can execute A+ expressions. This allows you to switch between A+ and C as needed. Note that you must start with an A+ process, however, to execute these dynamically loaded programs.

The entry points `pex()` and `ex()` allow you to do this. `pex()` takes one argument, a pointer to a string containing the A+ expression you wish to execute. `ex()` takes two arguments. The first is a context. The second is the string to execute in that context. The prototypes for these functions are:

```
I pex(I a);
I ex(CX c, C *s);
```

The longs returned by both functions are pointers to A+ objects.

Mapped Files

Mapped Files look very much like other A+ variables, from the C perspective, and have headers as described in "The Basic A+ Data Types". They have reference counts of 0 to distinguish them, however. That is, `(0==aobj->c)` means the object is a mapped file.

There is an entry point called `wr()` which will return 1 for a *writable* mapped file, and 0 otherwise. So a writable mapped file is indicated by `(0==aobj->c && wr(aobj))`.

If you write to an `aobj` where `(0==aobj->c && !wr(aobj))` you will cause a segv.

`wr()` references a variable called `wt`, which is a list of writable addresses. This code is in `y.c`, in the `a` source directory.

Memory Allocation in A+ - a Closer Look

A+ uses the function `ma()` to allocate memory. This function is specified to return memory locations that begin on 8-byte boundaries, freeing the last three bits for encoding purposes, which is how they are used.

A+ consists of several types of entities, all represented as integer-size objects. The last three bits of the object indicate the type of object.

The most common case (for our purposes) is for those three bits to be 000, which indicates a pointer to an A+ object. In this case, the pointer can be used as is.

Other codes require some manipulation. For example, if the code is 010, the object is a pointer to a symbol, and the last three bits must be cleared before the pointer is used. (As stated above, all pointers in A+ point to 8-byte boundaries, as allocated by `ma()`, so the last three bits must be 000, and it is this fact that allows A+ to use the last three bits for type encoding.)

Several macros are provided in `a/arthur.h` to query the type of an object. The next table gives a list of the macros, and the types of objects they represent.

Macros for Querying Object Type

Code	Macro	Object Type
0	`QA(a)`	pointer to A+ object (struct `a`)
1	`QV(a)`	global variable (struct `v`)
2	`QS(a)`	pointer to symbol (struct `s`)
3	`QE(a)`	pointer to expression (struct `e`)
4	`QN(a)`	flow-control/operator
5	`QL(a)`	local variable
6	`QP(a)`	primitive
7	`QX(a)`	dynamically loaded function

Also defined are macros which clear the last three bits for those entities which serve as pointers. They are:

`XS(a)` retrieve pointer to struct `s`
`XV(a)` retrieve pointer to struct `v`
`XE(a)` retrieve pointer to struct `e`

These three macros work by zeroing out the last three bits and casting the result to the appropriate pointer type. Notice that there are no macros for A+ objects, although you will occasionally have to cast them.

Finally, there are macros to add the proper code into the last three bits. They are:

`MV(a)` global variable (struct `v`)
`MS(a)` pointer to symbol (struct `s`)
`ME(a)` pointer to expression (struct `e`)
`MN(a)` flow-control structure or operator
`ML(a)` local variable
`MP(a)` primitive function
`MX(a)` dynamically loaded function

The Symbol Structure

The structure of symbols is defined in`a/arthur.h` as:

```
typedef struct s{struct s *s; C n[4];} *S;
```

`s` the next symbols. This field is included because all symbols created by A+ are stored in linked lists.

`n` a character string with the name of the symbol.

Symbols in Variables

Symbols are always contained in A+ objects of type `Et` (nested). In normal nested objects, the elements of `p[]` point to A+ objects (struct `a`). With symbols, they point to symbols (struct `s`).

Using Symbols in Functions

Recognizing symbols.

Whenever you encounter an object of type `Et`, the nested elements may be any type of A+ entity, or several types mixed together. You should always check the contents of `p[]` individually, using the `Q_()` macros described above, when using objects of type `Et`.

To check if an element of an `Et` object is a symbol, use the `QS()` macro.

Getting the name of a symbol.

To get the character string associated with the symbol (its "name"), you must turn the symbol into a pointer to an `s`-struct using the `XS()` macro, and access the `n` field within that structure.

Example 1: Recognizing a symbol and getting its name.

The following C function examines an A+ object and prints out the symbols it contains.

```
void printsymbols( aobj)
  A aobj;
{
  int i;
  S sym;
  if (Et != aobj->t ) {
    printf("object not nested\n");
    return;
  }
  for (i=0 ; i<aobj->n ; ++i ) {
    if (QS(aobj->p[i]) {
      sym = XS(aobj->p[i]);
      printf ("Symbol:%s\n", sym->n);
    } else printf("Not a symbol\n");
  }
}
```

Creating a symbol

It is an essential property of symbols that, if two symbols have the same name, they are the same symbol (point to the same memory location).

For this reason, you must always create symbols by using the `si()` function. This function takes a character string as an argument, and returns a pointer to an `s`-struct. If the symbol already exists, you get the current memory location. Otherwise, a new symbol is created and stored in A+, and the address of the new symbol is returned.

If you intend to insert a symbol into an A+ object, you must encode the last three bits as 010, which is best done with the `MS()` macro. Then load the symbol into the `p[]` field of an A+ object of type `Et`.

Example 2: Returning a symbol

The following function takes a string and returns a symbol with the string as the name.

```
A makesymbol(str)
  char *str;
{
  A res=gs(Et);
  res->p[0] = MS(si(str));
  return(res);
}
```

Comparing symbols

Because symbols with the same name are always in the same memory location, you don't need to use string comparisons to check symbols for identity. The result of `si()` will match for any symbol that you use. Just make sure that the symbols you are comparing either both have the 010 in their last three bits, or both have not.

Example 3: Comparing symbols.

The following function checks whether the A+ object contains the symbol `` `qwerty ``. If so, it returns 1, else 0.

```
queryqwerty( aobj)
  A aobj;
{
  int i;
  S qwerty = MS(si("qwerty"));     /* get symbol and set 010 code */
  if (Et != aobj->t ) return(0);
  for (i=0 ; i<aobj->n; ++i) {
      if (qwerty == (S) aobj->p[i]) return(1);
    }
  return(0);
}
```

13. Appendix: Miscellany

Importing and exporting data: *sys.exp* is monadic and takes any A+ data object as its argument, returning an encoded character vector. *sys.imp* is also monadic and takes any result of *sys.exp* as its argument and returns the decoded data object as its result. Results of *sys.exp* are convenient for writing data to files or sending it to another Unix process.

Bus error: The A+ interpreter attempted to read from an address outside your area, or an attempt was made to read or write without regard to address alignment. If a `⍕[error] : bus` (or a

`: bus`) error occurs, the application should be restarted. If the error can be recreated, please contact us to look into it. See `busexit.

Segv error: The A+ interpreter attempted to write in an address outside your area. If a `a[error] : segv` (or a `: segv`) error occurs, the application should also be restarted, just as for a bus error, and the error reported to the A+ development group, email id aplusdev. See `busexit.

Process `a' illegal instruction: This message is most commonly caused by a bus or segv error, and indicates that you continued your session after a bus or segv error occurred. The application should be restarted and, if you didn't continue after a bus or segv error, please contact us so that we can look into it.

If an error is signalled by a program, as opposed to occurring in a primitive function or operator, the stack variables are not likely to be helpful. In particular, if an error is signalled from within adap, the stack variables will be meaningless, since there is no point in adap to suspend at.

The system context function, `sys.getdomainname{}`, if successfully executed, returns a character vector containing the NIS domain name of the current host; otherwise, it returns Null.

14. Appendix: GNU Free Documentation License

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed

book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through

arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.