Walnut User System Documentation

Glen Pringle

December 6, 1995

Abstract

The Walnut User System provides an environment in which processes can be created and managed. It provides access to the console via GLLui, a screen and keyboard multiplexor. The Walnut shell utilizes a capability nameserver to allow the creation and manipulation of capabilities.
# Contents

1 GLui ........................... 1

1.1 Initialisation ................................. 1

1.1.1 The System-Wide Name Server Sets .............. 1

1.2 GLui Log Messages ........................... 3

1.3 User Commands ............................... 3

1.4 Messages ............................... 4

1.5 Creating new shells via login program .... 5

2 Walnut Login Program ......................... 6

2.1 Use ............................... 6

2.1.1 Starting a New Shell .......................... 6

2.1.2 Creating a New “User” ....................... 6

2.1.3 Editing Entry For a User ...................... 7

2.1.4 Running Under Unix .......................... 7

2.2 The Login Database ......................... 7

2.2.1 Format of the Login Database .............. 7

2.2.2 Finding the Database ...................... 8

3 The Shell ............................... 9

3.1 Job Control ................................. 9

3.2 Built-in Commands ........................... 10

3.2.1 alias ................................. 10

3.2.2 bind ................................. 10

3.2.3 clear/cls .............................. 11

3.2.4 cs ................................. 11
3.2.5 date ........................................ 11
3.2.6 dbg ....................................... 11
3.2.7 echo ....................................... 12
3.2.8 exit ....................................... 12
3.2.9 fill ....................................... 12
3.2.10 frun ..................................... 12
3.2.11 help/? ................................... 13
3.2.12 history / h ............................... 13
3.2.13 import .................................. 13
3.2.14 jobs ..................................... 13
3.2.15 kill ...................................... 14
3.2.16 mail ..................................... 14
3.2.17 pwd ...................................... 14
3.2.18 printenv ................................. 14
3.2.19 run ...................................... 14
3.2.20 set ....................................... 15
3.2.21 setenv ................................... 15
3.2.22 show ..................................... 15
3.2.23 sleep .................................... 16
3.2.24 unalias .................................. 16
3.2.25 unset .................................... 16
3.2.26 unsetenv ................................ 16
3.2.27 ver ...................................... 16
3.3 Shell Variables ................................ 17
3.3.1 Internal Format for Variables .......... 17
3.3.2 Special Variables ........................................ 18

3.4 Command Line Parsing ........................................ 19
  3.4.1 Shell Variables ........................................ 19
  3.4.2 Nameserver Bindings ................................... 19
  3.4.3 History Events ......................................... 20
  3.4.4 Alias Substitutions .................................... 20

4 Nameserver 21
  4.1 Construction of the Nameserver .............................. 21
    4.1.1 Bindings ........................................... 21
    4.1.2 Sets ............................................... 22
    4.1.3 Included sets ....................................... 22
  4.2 Internals ................................................ 23
  4.3 Programmers Interface .................................... 23
    4.3.1 Finding the Value of a Binding ....................... 24
    4.3.2 Setting the Value of a Binding ...................... 25
    4.3.3 Initializing Nameserver .............................. 25
    4.3.4 Miscellaneous ....................................... 25

5 Standard Utilities ........................................ 26
  5.1 cat ...................................................... 26
  5.2 drive .................................................... 26
  5.3 finstall .................................................. 26
  5.4 ls - List Set ............................................. 27
  5.5 mcopy ..................................................... 27
  5.6 mkset .................................................... 28
5.7 ren .................................................. 28
5.8 time .................................................. 28
5.9 unbind ............................................... 28
5.10 wyrm ............................................... 29
1 GLui

GLui (GLen’s user interface) is a screen multiplexer, providing up to ten virtual terminals under walnut. An integral part of the system, it requires low level access to the screen hardware and keyboard buffer. A user at the console can control access to input/output through GLui commands.

1.1 Initialisation

Aside from initialisation of variables etc, during the initialisation phase GLui waits to receive capabilities giving access to the screen and keyboard, as well as other resources. These are sent by initproc and are listed in table 1. GLui requires all these capabilities and will not continue until they have all been received.

<table>
<thead>
<tr>
<th>Resource Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminal emulator buffer</td>
<td>screen cursor movement</td>
</tr>
<tr>
<td>keyboard buffer</td>
<td>keyboard input buffer</td>
</tr>
<tr>
<td>screen memory</td>
<td>screen (console) memory</td>
</tr>
<tr>
<td>the Wall graffiti</td>
<td>write capability for the Wall</td>
</tr>
</tbody>
</table>

Table 1: Resources sent by initproc to GLui

Any process requiring access to the screen and keyboard must be able to contact GLui to obtain this access. This is accomplished through the use of messages sent to GLui by process requiring this access. GLui writes a mail capability into the Wall, as detailed in table 2.

<table>
<thead>
<tr>
<th>Word in the Wall</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>theWall[0]</td>
<td>GLUL MAGIC</td>
<td>Magic number (value 0x4200000d)</td>
</tr>
<tr>
<td>theWall[1]</td>
<td>vol</td>
<td>volume of GLui mail capability</td>
</tr>
<tr>
<td>theWall[2]</td>
<td>serial</td>
<td>serial number of GLui mail capability</td>
</tr>
<tr>
<td>theWall[3]</td>
<td>pass1</td>
<td>password 1 of GLui mail capability</td>
</tr>
<tr>
<td>theWall[4]</td>
<td>pass2</td>
<td>password 2 of GLui mail capability</td>
</tr>
</tbody>
</table>

Table 2: GLui’s mail name in the wall

1.1.1 The System-Wide Name Server Sets

During the creation of the walnut system under unix (using the standalone version of drive), an object is created in which names of programs and “files” are put,
which are to be available to all processes on the system. Currently these are
categorised into /bin and /lib sets.

GLUi has the name of the database\(^1\) which contains objects to be placed in the
system wide /bin and /lib sets hard coded in as listed in figure 1.

```
#define NS_DATABASE_VOL   0x8888
#define NS_DATABASE_SERIAL 0x16c
#define NS_DATABASE_PASS1  0x8000406
#define NS_DATABASE_PASS2  0x407
```

**Figure 1:** Name of database for objects which are to be added to the system-wide
name server set

Entries in the database are in one of two formats. The first is for names of
processes to be added to the /bin set, and consists of the name of the process
and two capabilities (the first being the text (code) object for the program and
the second being the data object). The second type of entry in the database is
for objects to be added to the /lib set, and consists of the name (binding) of the
object followed by the capability for it.

**Figure 2** contains the data structure for the entries. The end of the list is indicated
by a null entry (all the fields in the entry are zero).

```c
typedef struct ProcessInfo {
    char name[32];    /* Name of the program */
    Cap1 text;        /* Text object of program */
    Cap1 data;        /* Data object of program */
} processInfo;
```

**Figure 2:** Data structure for entries in the database of programs

GLUi creates a new name server set which has two sets in it - the /bin and /lib
sets. Bindings for entries in the database are added to these two sets. If an entry
is a process, with name *prog* in the database, two bindings are added to the /bin
set:

- *prog.cbn* is the capability of the code for the program

\(^1\)This database is not a name server database object.
• *prog.dbn* is the capability of the data for the program

For entries which are not processes, a binding (given by the name of the object as found in the database entry) to the capability found in the entry is made in the */lib* set.

The capability and offset of the name server set is available in the Wall by GLui once it has created it. This is show in table 3.

<table>
<thead>
<tr>
<th>Word in the Wall</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>theWall[5]</td>
<td>NS_SET</td>
<td>Magic number (value 0x22222222)</td>
</tr>
<tr>
<td>theWall[6]</td>
<td>vol</td>
<td>volume of set</td>
</tr>
<tr>
<td>theWall[7]</td>
<td>serial</td>
<td>serial number of name server set</td>
</tr>
<tr>
<td>theWall[8]</td>
<td>pass1</td>
<td>password 1 of name server set</td>
</tr>
<tr>
<td>theWall[9]</td>
<td>pass2</td>
<td>password 2 of name server set</td>
</tr>
<tr>
<td>theWall[10]</td>
<td>offset</td>
<td>offset of name server set in object</td>
</tr>
</tbody>
</table>

Table 3: Entry in the Wall for system-wide name server set

There is one object found in the database which is not added to the system-wide name server set. That is the password database for the login program. GLui stores the capability for this object which is passed to the login program when it is run (see section 1.5).

### 1.2 GLui Log Messages

A mechanism is provided for GLui to log events. This consists of a function call, *print1()* , analogous to *printf()* . Printing log messages results in the messages being added to an object which is in the same format as the screen memory, thus allowing it to be copied directly into screen memory when the logs are to be displayed. ANSI colour codes may be used to set the colour to be printed in, with the definitions in figure 3 found in the header file, *log.h*.

### 1.3 User Commands

Control of GLui by the user at the console is accomplished through various key commands which are prefixed by the key, *control-a*. This puts GLui into command mode, whereby the next key is interpreted as a command rather than being sent to the process attached to the current active screen. The commands available are listed in table 4. Any other keys are ignored.
#define NORMALMODE  "\033[0m"
#define BOLDMODE    "\033[1m"
#define BLINKMODE   "\033[5m"
#define REVERSEMODE "\033[7m"
#define INVISIBLE   "\033[8m"
#define BLACKFG     "\033[30m"
#define REDFG       "\033[31m"
#define GREENFG     "\033[32m"
#define YELLOWFG    "\033[33m"
#define BLUEFG      "\033[34m"
#define MAGENTAFG   "\033[35m"
#define CYANFG      "\033[36m"
#define WHITEFG     "\033[37m"
#define BLACKBG     "\033[40m"
#define REDBG       "\033[41m"
#define GREENBG     "\033[42m"
#define YELLOWBG    "\033[43m"
#define BLUEBG      "\033[44m"
#define MAGENTABG   "\033[45m"
#define CYANBG      "\033[46m"
#define WHITEBG     "\033[47m"

Figure 3: ANSI Formatting and Colour codes

1.4 Messages

Following the initialisation, GLui enters an endless loop where it waits for requests and processes them. Each time through the loop the following is performed:

- Check for magic number in first word of the Wall. If it isn’t present, there must have been a system shutdown which erases the Wall, so rewrite values. Send startup messages to any processes we know about.
- Check if there’s any output for the current screen. If so, process it (or up to 80 characters). This involves writing it to the console screen.
- Check for messages. These messages are listed in table 5.
- If there is any keyboard input, process it. This involves checking for the command character (control-a), and if it is pressed, entering command mode, where the following character will be interpreted as a command to GLui (eg, to change to another virtual screen).
<table>
<thead>
<tr>
<th>Command</th>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR_META</td>
<td>‘a’</td>
<td>Send the screen command character (control-a) to the currently active screen</td>
</tr>
<tr>
<td>SCR_OTHER</td>
<td>control-a</td>
<td>Toggle to other screen</td>
</tr>
<tr>
<td>SCR_NEXT</td>
<td>space</td>
<td>Next screen</td>
</tr>
<tr>
<td>SCR_TIME</td>
<td>‘t’</td>
<td>Current time (toggle)</td>
</tr>
<tr>
<td>SCR_KILL</td>
<td>‘K’</td>
<td>Kill current screen</td>
</tr>
<tr>
<td>SCR_CREATE</td>
<td>‘c’</td>
<td>Create a new shell</td>
</tr>
<tr>
<td>SCR_LOG</td>
<td>‘?’</td>
<td>Display GLui’s log file</td>
</tr>
<tr>
<td>SCR_HELP</td>
<td>‘?’</td>
<td>Help for GLui</td>
</tr>
</tbody>
</table>

Table 4: User Commands in GLui

<table>
<thead>
<tr>
<th>Message</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE_REQ_KEYBOARD_SCREEN_CAP</td>
<td>0x17081968</td>
<td>Request a screen, and when it is created, switch to it</td>
</tr>
<tr>
<td>REQUEST_KEYBOARD_SCREEN_CAP</td>
<td>0x17081969</td>
<td>Request a screen</td>
</tr>
<tr>
<td>REQUEST_KEYBOARD_SCREEN_MEM</td>
<td>0x1708196b</td>
<td>Request screen memory</td>
</tr>
<tr>
<td>DONE_KEYBOARD_SCREEN_CAP</td>
<td>0x1708196a</td>
<td>Process finished with screen</td>
</tr>
<tr>
<td>GLUECHO_ON</td>
<td>0x1708196c</td>
<td>Turn echo on for screen</td>
</tr>
<tr>
<td>GLUECHO_OFF</td>
<td>0x1708196d</td>
<td>Turn echo off for screen</td>
</tr>
</tbody>
</table>

Table 5: Messages GLui can interpret

1.5 Creating new shells via login program

When the user enters the SCR_CREATE command (see section 1.3), GLui creates a new login process, passing it the name of the password database. This program is detailed in section 2.
2 Walnut Login Program

The walnut login program is invoked by the user through using `control-a c` in GLui. It allows the user to enter a login (user) name and password to start a shell. The login program has a database containing the username, password and nameserver set. A successfully entered username and password will allow a new shell to be started which is passed the nameserver set which is in the database.

2.1 Use

2.1.1 Starting a New Shell

The login program prompts the user for a login/user name:

```
login:
```

After a name is entered, the login program does a lookup in the database to see if there is an entry for such a user. If there is, a password must be entered (note the password is not echoed to the screen):

```
Password:
```

If the password matches, then a shell is started which is given the capability to the nameserver set object found in the login database. Money is taken from the nameserver set object for creating the shell process. Thus the nameserver set object must have enough money in it a shell to be run.

2.1.2 Creating a New “User”

If `add` is entered at the `login:` prompt, the login program asks if the user wishes to create a user with that name. If the user answers ‘y’, then a password must be supplied (it must be entered twice to verify it) and then the nameserver set (consisting of a capability to an object and offset) must be entered. Note that a null password may be entered. In this case, the login program will not prompt for a password after the username has been typed in.

Under walnut, the login program will verify the nameserver set’s object by attempting to withdraw some money (one “razoo”) from it. If this fails, then the user won’t be created. If it is successful and no such user already exists, then
the user will be created. The reason for deducting the money is so that brute force attempts to find out valid usernames will not be feasible since there is a cost associated with every failure.

2.1.3 Editing Entry For a User

Entering edit at the login: prompt will allow the database entry for a user to be edited. The login program prompts for username and password and if they are successfully entered a short menu is brought up which allows the password to be changed, the whole entry to be deleted from the database and the nameserver set to be changed. If the delete option is selected, the password must be correctly entered before the entry is deleted. This doesn’t apply if the user has a null password.

When editing has been completed, the quit (‘q’) option will save the changes and quit editing, while the abort (‘a’) option will abort any changes made before quitting editing.

2.1.4 Running Under Unix

The login program may be run under unix. It uses the file called passwd in the current working directory as the database file. Any changes to the database will be saved in this file when the login program is exited from (using EOF at the login prompt). The passwd file in the directory ~/walnut/multi/prog/login is used in the system for the database. Thus users may be created in this database which exist when the walnut system is run.

2.2 The Login Database

The login program uses a simple database to store details of usernames, passwords and nameserver sets. The database object in Walnut must have type equal to PASSWD_TYPE, as define in the include file, passwd.h:

#define PASSWD_TYPE 0x777

2.2.1 Format of the Login Database

The database is composed of entries each 64 bytes long. The first entry is a header which is as follows:
typedef struct _PasswdHeader {
    Uw magic;       /* Magic number */
    Uw numberEntries; /* Number of entries in password object */
    Uw objectSize;   /* Size of password object */
    Uw lock;         /* Write lock on object */
} passwdHeader;

The magic number PASSWD_OBJECT_MAGIC is defined as:

#define PASSWD_OBJECT_MAGIC 0x73736150

Each entry within the database has the following format:

typedef struct _PasswdEntry {
    char   username[16];   /* username */
    char   password[16];   /* encrypted password */
    Capl   setCap;         /* nameserver database set object */
    Uw     setOffset;      /* offset into database object for set */
    Uw     fill1;          /* Filler word to bring to 64 bytes */
    Uw     fill2;          /* Filler word to bring to 64 bytes */
    Uw     fill3;          /* Filler word to bring to 64 bytes */
} passwdEntry;

The entries in the database are not sorted. New entries are added to the end after the last entry. Deletion of entries may rearrange the order of the remaining entries.

2.2.2 Finding the Database

When run under unix, the login program has the location of the database file compiled in using the macro PASSWDFILE (defined in the makefile, or failing that, as "./passwd"). Under walnut, however, GLui must give the capability of the object containing the database to the login program. This is accomplished by passing it in through the command line arguments.

Under walnut, the database is passed in as the command line arguments (ie, argv[] in main()). The login program is generally created by GLui who keeps the capability for the login database.
3 The Shell

The walnut shell provides an environment for running new processes and manipulating capabilities via the nameserver (Section 4). Although it is currently much less powerful than Unix’s csh/tcsh, it is hoped that it will be developed similar to tcsh.

When run, the shell checks to see if it has stdin and stdout already set up. If not, it sends a message to GLUi requesting input/output buffers and sets these up as stdin and stdout. A new nameserver database is created for the shell, and associated initialisation occurs.

The shell enters an infinite loop where mail messages are interpreted, keyboard input is parsed and the resulting commands are executed. When input from the keyboard is received, the shell passes it to the function addToLineO which adds it to the current line. This function allows command line editing (currently restricted to the use of the backspace key) and when the enter key is pressed, calls the appropriate functions to the parse the line and execute the command.

3.1 Job Control

The shell allows the user to create new processes, and maintain limited control over their execution. Processes may be started either through the run command (see section 3.2.19) or by simply entering the name of a program which is present in the current set or the /bin set (Section 1.1.1).

The run command requests a new screen from GLUi and uses it to set up stdin and stdout for the new process. Currently, the parameters passed to the new process are those typed in at the command line.

When a new process in the /bin set is started by entering it’s name, the shell “lends” its stdin and stdout to the new process and also passes argc and argv to the new process.

The set ./process is used to store details on running processes controlled by the shell. ./process/lastProcess is an integer indicating the job number of the last process. Job numbers are a unique number allocated by the shell to each running process. They are allocated in ascending order starting at one (1). When a new process is started, ./process/lastProcess is incremented and the new process is giving the job number ./process/lastProcess. If we call this job number x, a new set ./process/x is created and information relating to job x is stored in this set. This information includes the capability of the process and related capabilities.
(such as the text and data objects).

The shell regularly checks each job to see if the process is still alive (by sending a COOEE message to the process’s subprocess zero). If it has died, the shell will delete any objects created specifically for the process and finally delete the set "/.process/lastProcess/".

Job numbers are unique across all shells sharing the same nameserver sets. This can lead to problems since shells do not remember if they were the shell which created a process, so all shells will watch for a given process to die. It is hoped that ultimately, the names of processes a shell has created will be stored internally by the shell. The details of running processes will be placed in a set called "/.process/vol.serial", where vol and serial are respectively the volume and serial numbers of the running process. This will fix the problem of multiple shells watching for dead processes.

### 3.2 Built-in Commands

The shell has several built-in commands which are executed directly by the shell, without the overhead of creating a new process. These are briefly documented below.

#### 3.2.1 alias

Usage: alias
  alias name
  alias name wordlist

See section 3.4.4 for more details.

#### 3.2.2 bind

Usage: bind
  bind name
  bind name integer
  bind name string
  bind name vol serial pass1 pass2
  bind name [-i] vol serial pass1 pass2 offset
3.2.3 clear/cls

Usage: clear [debug]
       cls

These commands clear the screen. This is accomplished by printing the string
"^[H^[J" to the standard output.

If the debug option is used with the clear command, then the debug screen will
be cleared.

3.2.4 cs

Usage: cs [nameserver set]

Changes the current working set to the nameserver set specified. If none is given,
then changes to the root set, "/". This command is analogous to unix’s cd(1)
command.

3.2.5 date

Usage: date

Print out the current date and time.

3.2.6 dbg

Usage: dbg lineNumberOffset command [options]

Executes the command, but when it finishes, extracts the number of the last
linenumber executed from the data object at the given offset. lineNumberOffset
is a hex number.

Not available in unix version.
3.2.7 echo

Usage: echo string

Write the given string to the standard output.

3.2.8 exit

Usage: exit

Exit the shell. If there are still running jobs, don’t exit unless previous command was also exit.

3.2.9 fill

Usage: fill

The fill command is a debug one which prints 10,000 characters to the standard output. These are printed using the following algorithm.

for i:=1 to 10000
    print "%c", i%64 + 32

3.2.10 frun

Usage: frun [options]

Options:  -m money  How much money to give new process
          -f fname   Specifies the name of the program to run off the floppy disk. Two files are loaded with this name with the extensions .cbn for the code binary, and .dbn for the data binary.
          -s cod dat Use the files from the floppy cod for the text(code) and dat for the data.
          -n name    Give name to new process
Copies files off disk containing code and data objects then creates a new process using these objects.

Not available in unix version.

3.2.11 help/?

Usage: help
    ?

Print out help on several shell commands.

3.2.12 history / h

Usage: history
    h

Print out command history.

3.2.13 import

Usage: import nameserver_file [binding]

Imports the specified unix_file into the walnut system and if binding is specified, adds a nameserver binding for it into the nameserver set.

Note that if the walnut system is rebuilt, the file will have to be re-imported (ie, any bindings in the nameserver will be stale).

Only available in unix version.

3.2.14 jobs

Usage: jobs

Print out currently running jobs

Not available in unix version.
3.2.15  kill

Usage: kill %x1 [%x2 %x3 ...]

Kills the running job[s] numbered x1, x2, ....
Not available in Unix version.

3.2.16  mail

Usage: %s cap1 subpn [options]
Options:  -m n   Send n money with message (money is a
decimal number)
          -a     Set ascii text mode - input is ascii text

Send a mail message to a process. cap1 is the capability of the process (vol serial pass1 pass2)
and subpn is the subprocess number. All values are hexadecimal.
Not available in Unix version.

3.2.17  pwd

Usage: pwd

Print the current working set’s path.

3.2.18  printenv

Usage: printenv [name]

Prints out the names and values of the environment variable name, or all the
environment variables if none specified.

3.2.19  run

Usage: run textCapability dataCapability [options]
Options:  -m money   How much money to give new process
-t  Don’t copy text object; use capability given
-d  Don’t copy data object; use capability given
-n name  Give name (in nameserver) to capability of new process

Creates a new process, using the two capabilities (specified by the hexadecimal values of the volume, serial number and two passwords) of the text and data objects to copy.

Not available in unix version.

3.2.20 set

Usage: set
    set name
    set name=word
    set name=(wordlist)

Works the same way that the set command in unix’s csh(1) does, setting the shell variable name to the specified word or wordlist. See section 3.3 for more details.

3.2.21 setenv

Usage: setenv
    setenv name value
    setenv name

Setenv without any parameters will list the values of all environment variables (equivalent to printenv (see section 3.2.18 without any arguments).

The following forms are not yet implemented. Specifying a name and value will set the environment variable name to value. If value is not specified, then name will be set to an empty string.

3.2.22 show

Usage: show
show name

3.2.23 sleep

Usage: sleep sleeptime

Go to sleep for the specified number of seconds, or if -1 and under the walnut system, then go to sleep until a message is sent to us.

3.2.24 unalias

Usage: unalias name[s]

See section 3.4.4 for more details.

3.2.25 unset

Usage: unset name[s]

Remove the specified shell variable[s]. See section 3.3 for more details.

3.2.26 unsetenv

Usage: unsetenv [name]

Not yet implemented.

3.2.27 ver

Usage: ver

Print out the date the current version of the shell was compiled
3.3 Shell Variables

There are available to the user, variables which may be assigned string values. These values are either a word or wordlist, and are assigned using the set command (section 3.2.20). The unset command (section 3.2.25) will remove the named shell variable/variables.

3.3.1 Internal Format for Variables

Shell variables are stored in a linked list as shown in figure 3.3.1. Each entry in the list is of the type setaliasData as defined by the following.

```c
typedef struct _setalias_Data {
    setaliasContents    contents; /* The data */
    struct _setalias_Data *next;    /* Next item in linked list */
} setaliasData;
```

The structure setaliasContents contains the name, which is the name of the shell variable, type which is the type of the variable (if it is a word or a wordlist) and the actual value of the variable, data.

```c
typedef struct _setalias_Contents {
    char     *name;    /* Name of the variable/alias */
    int      type;     /* WORD or WORDLIST */
    unionVal data;     /* The value of the variable/alias */
} setaliasContents;
```

The values which type may have are as follows:

```c
#define WORD 1
#define WORDLIST 2
```

If the type of the data is a word, then data is a pointer to a string; if it is a wordlist, then data is a pointer to an array of pointers to strings. This array is null-terminated.

```c
typedef union UnionVal {
    char    *word;
    char    **wordlist;
} unionVal;
```
3.3.2 Special Variables

The following variables have special meaning to the shell or are set by the shell.

- **prompt** A string which contains the prompt for the shell.
- **user** The name of the current “user”, if it is known. This is taken from the environment variable **USER**.
- **fg** The foreground colour of the screen. This may contain a numeric value from 0 to 7, or a string description from the following set: black red green yellow blue magenta cyan white
- **bg** The background colour, in same format as the foreground colour.
- **blink** If set, blink attribute is set for text.
- **reverse** If set, text colour is reversed.
- **bold** If set, text is bold.
- **diag** Setting different bits in this integer variable will cause debug messages to be displayed on the debug screen. Table 6 describes briefly the effect of setting different values in this variable.
- **cws** Current working set - path to the current working set
- **database** Nameserver database capability and offset
- **version** Version of the shell, currently the compilation time
<table>
<thead>
<tr>
<th>Diagnostic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAG_GENERAL</td>
<td>0x0001</td>
<td>General diagnostic messages</td>
</tr>
<tr>
<td>DIAG_MESSAGES</td>
<td>0x0002</td>
<td>Mail messages</td>
</tr>
<tr>
<td>DIAG_ERRORS</td>
<td>0x0004</td>
<td>Errors</td>
</tr>
<tr>
<td>DIAG_HEARTBEAT</td>
<td>0x0008</td>
<td>Display “heartbeat”</td>
</tr>
<tr>
<td>DIAG_TEMP</td>
<td>0x0010</td>
<td>Temporary, for debugging</td>
</tr>
<tr>
<td>DIAG_PROGS</td>
<td>0x0020</td>
<td>Creation of processes</td>
</tr>
</tbody>
</table>

Table 6: Values of diag variable

### 3.4 Command Line Parsing

The shell parses the command line before interpreting it and passing it to the process to be executed. This process includes substitution of shell variables and nameserver bindings, history expansion and alias substitution. These processes are detailed in the following sections.

#### 3.4.1 Shell Variables

The dollar sign (“$”) is used to prefix shell variable names which are expanded by the shell. For example, to print the current working set, the following command might be used (this is equivalent to using the command, `pwd`).

```
walnut> echo $cws
/bin
walnut>
```

#### 3.4.2 Nameserver Bindings

Prefixing a nameserver binding with the at sign (“@”) will result in the shell performing an expansion off the command line, with the at sign and binding name being replaced with the value of the binding. The following command will display the capability of the root nameserver set.

```
walnut> echo @/ 
8888 2e8 8000040c 40d 80 
walnut>
```
3.4.3 History Events

History expansion is available using the exclamation mark (“!”) in a similar manner to unix’s `csh(1)`. Currently only a subset of `csh(1)`’s syntax is available.

A double exclamation mark (“!!”) will be replaced by the shell with the previous command, or an exclamation mark followed by a number will cause a replacement by the numbered history event.

3.4.4 Alias Substitutions

Aliases, as used in the unix `csh(1)`, are provided in the walnut shell. They are implemented with the same code as shell variables (section 3.3.1).
4 Nameserver

Objects in the walnut kernel are identified by capabilities which consist of four words each of length 32 bits. Access to objects is via these capabilities, which are very unwieldy for humans to use. The nameserver allows people to give names to capabilities, thus allowing them to use the name to reference a capability and transferring the work of manipulating the four 32 bit numbers from the human to the machine.

The nameserver allows the creation of a binding from a name to a capability. This name may be a string which is selected by a human. Bindings are grouped together in sets with the sets of bindings stored in objects, so that a person only needs to remember one capability - that of the object containing their main set of nameserver bindings.

4.1 Construction of the Nameserver

4.1.1 Bindings

The nameserver has at its lowest level bindings which are a string (the name of the binding), the type of the binding and what the name is bound to. This is illustrated in figure refig:binding. The name field is the name of the binding and must be less than eighty characters long. The bind field is the value of the binding which is five (32 bit) words long, and is interpreted according to the type field. The values for this field are listed in table 7.

```c
typedef struct Binding {
    Uw    type;
    Uw    next;
    Uw    previous;
    char  name[80];
    unionBinding  bind;
} binding;
```

Figure 5: Struct for a binding

The type unionBinding in figure 5 is a C union of the four types shown in figure 6.
<table>
<thead>
<tr>
<th>Binding Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE_OBJECT</td>
<td>0x70000000</td>
<td>Object is a “file”</td>
</tr>
<tr>
<td>CODE_OBJECT</td>
<td>0x70000001</td>
<td>Object contains process code</td>
</tr>
<tr>
<td>PROCESS_OBJECT</td>
<td>0x70000002</td>
<td>Object is a process</td>
</tr>
<tr>
<td>DATA_OBJECT</td>
<td>0x70000003</td>
<td>Object contains data for a process</td>
</tr>
<tr>
<td>SET_OBJECT</td>
<td>0x70000004</td>
<td>Binding is the name of a set</td>
</tr>
<tr>
<td>INCLUDED_SET_OBJECT</td>
<td>0x70000005</td>
<td>Binding to an included set</td>
</tr>
<tr>
<td>INTEGER_VALUE</td>
<td>0x70000006</td>
<td>Binding is an integer value</td>
</tr>
<tr>
<td>STRING_VALUE</td>
<td>0x70000007</td>
<td>Up to 20 character string</td>
</tr>
<tr>
<td>LONG_STRING</td>
<td>0x70000008</td>
<td>String longer than 20 chars (Not implemented)</td>
</tr>
</tbody>
</table>

Table 7: Types of Nameserver Bindings

### 4.1.2 Sets

Bindings are grouped together into sets. These sets are not the mathematical sets, obeying set theory. A user may have several different sets of bindings, and they will use one of them as their current working set of bindings. All nameserver lookups are done in this current working set. Some of the bindings in the current working set may be bindings for other sets, which allows the user to access bindings in other sets. This feature has been developed in such a way that the nameserver sets can be used in a way analogous to the hierarchical directories in most operating systems. However, they are still structurally very different as can be seen in figure 7.

A user will typically have a main or root nameserver set and bindings in that set for other sets of bindings which they have. The slash character (“/”) is used as a separator of sets, with the root set named `/`. So for example, to refer to the set which is on the bottom of figure 7, one could use `/s/u` or alternatively, `/a/r/u`. This provides an environment very similar to that in unix where slashes are used to separate directories but it must be remembered that this is not a tree-structured system.

### 4.1.3 Included sets

Often one may wish to add the bindings in one set to those in another set. For example, if a set contained bindings for all of a users programs, and another set contained the bindings for system-wide programs, the user may wish to include the system-wide programs into his personal set of bindings. This can be accomplished using an included set. A binding is made to the set to be included, and
typedef struct obj_binding {
    Cap1 cap;
    Uw offset;
} objBinding;

typedef struct val_binding {
    Uw value[5];
} valBinding;

typedef struct str_binding {
    char value[5*4];
} strBinding;

typedef struct int_binding {
    Sw value;
    Uw dum[4];
} intBinding;

Figure 6: Definition of unionBinding

designed to be an included set. When the nameserver searches the set for a binding, if it isn’t found in the set being searched, then all included sets are searched (and any included sets they may contain) until a matching binding is found.

4.2 Internals

Linked list used for sets. Stored in an object, with cap1 to object and offset in object to identify the set.

4.3 Programmers Interface

Although lower level nameserver subroutines don’t require one, all standard nameserver subroutines require as a parameter a nameserverDB struct, as shown in figure 8. This contains the capability and offset for the root set in the fields rootObj and rootOffset, with the field rootLoaded storing the memory location where the object is loaded, if it is loaded into this process’s address space. Similarly, the fields cwsObj, cwsOffset and cwsLoaded contain the details of the
current working set. The path is the list of bindings relative to the root set which will lead to the current working set.

4.3.1 Finding the Value of a Binding

There are three functions related to finding the value of a binding. These are `getData`, `getCapName` and `getInt` which are described below.

```c
int getData(nameserverDB *db, char *name, Uw *type, Uw *b);
```

`GetData()` retrieves the value for the binding name from the database db. The type of this binding is returned in the variable type and the value in b.

```c
int getCapName(nameserverDB *db, char *name, Cap1 *cap);
```

`GetCapName()`

```c
int getInt(nameserverDB *db, char *name, int *value);
```
typedef struct NameServerDB {
    Capl  rootObj;
    Uw    rootOffset;
    Uw    rootLoaded;
    Capl  cwsObj;
    Uw    cwsOffset;
    Uw    cwsLoaded;
    char  path[1024];
} nameserverDB;

Figure 8: Struct for a nameserver database

4.3.2 Setting the Value of a Binding

int setName(nameserverDB *db, char *name, Uw type, Uw *b, int replace);
int setCapName(nameserverDB *db, char *name, Uw type, Capl *cap, int replace);
int setInt(nameserverDB *db, char *name, int value, int replace);

4.3.3 Initializing Nameserver

int initSets(nameserverDB *db); void initDB(nameserverDB *db, char **envp);

4.3.4 Miscellaneous

void mkset(nameserverDB *db, char *name); int rmset(nameserverDB *db, char *name);
void cs(nameserverDB *db, char *name); char *getPath(nameserverDB *db);
char *path2set(nameserverDB *db, char *p, Capl *cap, Uw *offset);
5 Standard Utilities

Most of these utilities are implementations of standard unix utilities, adapted to
the walnut system. Instead of files, objects will be used for many utilities, and
nameserver sets replace directories.

5.1 cat

Usage: cat [capability] ["file"]

Read the named file (or object) and print its contents on standard output. This
utility is named cat for its similarity to unix’s cat(1). It currently does not
concatenate files.

Cat does not print out characters which have the value zero. This is due to the
fact that if an object contains text, zeros are generally found as fillers to the
end of the object. There are very few occasions when one will want to print out
these zeros if the use of the cat utility is to display the contents on the screen.
However, this utility is no compatible witht eh unix cat(1), so it needs to be
re-written at some stage.

5.2 drive

Usage: drive

Drive provides an interface to the kernel system calls. It has no documentation.

5.3 finstall

Usage: finstall [options] [program_name]

Options: -b install_name Set binding for installed program to install_name

The finstall program allows a program to be installed from the floppy drive. It
copies the files program_name.cbn and program_name.dbm from the floppy into
the current working set (adding bindings of the same name unless the -b option
is used). If bindings with the same names already exist in the current working
set, they will be deleted, and the objects they reference will be deleted.
5.4 ls - List Set

The ls command is analogous to unix’s ls(1) command, however under walnuts lists sets and bindings, rather than listing directory contents (since Walnut doesn’t have directories and files as such). It has a similar usage to ls(1):

Usage: ls [options] [bindings]

Options:  -l List one binding per line
          -F Put a slash (‘/’) after bindings which are sets
          -a List bindings beginning with a dot (‘.’)
          -l Long listing - list type and value of binding
          -r Reverse order of listing

Running ls with no command line arguments will list the names of the bindings in the current working set (this does not list the values of the bindings). Specifying a binding (or bindings) will cause the names of the bindings to be listed if they are not sets or the bindings within that set to be listed if they are sets; the -l option is required to list the value of the binding.

Bindings which begin with a dot (“.”) are not listed by default. The -a option is required to list these bindings.

5.5 mcopy

Usage: mcopy filename [-t type] [-l limit] [-n namebinding]

Options:  -t Type (in hexadecimal) given to the object created
          -l Limit (in hexadecimal) given to the object created
          -n The name for the binding added to the nameserver for object created.

Mcopy will copy the file filename from the floppy disk drive, creating an object to store the contents of the file in.

The file on the floppy must be in the root directory as sub-directories are not yet implemented.
5.6 mkset

Usage: mkset [-op] set_name[s]

Options:  -o  Create a new object for the new set  
           -p  First create all ‘parent’ sets which don’t exist

The `mkset` command is analogous to unix’s `mkdir(1)` command. It creates a new nameserver set and adds a binding for it to the nameserver.

Note that the `-p` option is not yet implemented.

5.7 ren

Usage: ren oldbinding newbinding

Renames the nameserver binding `oldbinding` to `newbinding`. This command is similar in concept to unix’s `mv(1)` command.

5.8 time

Usage: time command [arguments]

Time times the execution of the specified program, `command`. This program is invoked by the `time` command as though it had been run from the command line as “`command [arguments]`”.

5.9 unbind

Usage: unbind binding[s]

The specified nameserver binding[s] are removed. This does *not* remove the object associated with the binding.
5.10 wyrm

Usage: wyrm [speed]

Wyrm is a simple game, whereby the user controls a worm which moves around
the screen. The aim is to gather as much money as possible. Money consists
of yellow digits which are randomly placed around the screen. When a digit is
eaten, that amount is added to the score and the size of the worm grows by that
amount. The game is ended when the worm runs into the wall or part of it’s own
body.

The speed can be varied from 1 (very slow) to 9 (very fast). The speed can be
specified as a command line parameter, or it can be changed during the game
using the number keys.

A high score table is kept, with the top 16 high scores. They are stored in the file
/lib/wyrm.score. This table can be viewed anytime while running wyrm using
the “t” key.

Help, including movement keys, is available from the question mark key (“?”).