DRVBLD: A UNIX DEVICE DRIVER BUILDER

THESIS

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Agustin F. Cano, B.S.
Denton, Texas
May, 1992
Cano, Agustin F., **DRVBLD: A Unix Device Driver Builder.**

Master of Science (Computer Science), May 92, 49 pp., 1 table, 10 illustrations, 3 appendices, bibliography, 6 titles.

New peripheral devices are being developed at an ever increasing rate. Before such accessories can be used in the UNIX environment (UNIX is a trademark of Bell Laboratories), they must be able to communicate with the operating system. This involves writing a device driver for each device. In order to do this, very detailed knowledge is required of both the device to be integrated and the version of UNIX to which it will be attached. The process is long, detailed and prone to subtle problems and errors. This paper presents a menu-driven utility designed to simplify and accelerate the design and implementation of UNIX device drivers by freeing developers from many of the implementation specific low-level details.
Copyright by
Agustín F. Cano
1992
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vii</td>
</tr>
</tbody>
</table>

## Chapter

1. INTRODUCTION ........................................ 1
2. OVERVIEW ........................................... 2

### Configuration
- Adapting to the host system
- Driver-related information
  - General information
  - Driver-specific items
- The User Interface
- The Driver Generator
  - Generating the device driver code
  - Generating the Makefile and other scripts
- The DRVBLD directory tree
- *.SH files

3. DESIGN AND IMPLEMENTATION ....................... 16

### The User Interface
### The Driver Generator
- Files used and generated
  - Drivers
  - Code
  - Data
- How the code is generated
  - Scan of *.h files
  - Information gathered at installation time
  - User input
- Executing scripts and generating code
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. SIMPLE EXAMPLES</td>
<td>32</td>
</tr>
<tr>
<td>5. SUMMARY AND CONCLUSIONS</td>
<td>35</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>37</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>40</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>46</td>
</tr>
<tr>
<td>REFERENCE WORKS</td>
<td>49</td>
</tr>
<tr>
<td>Table</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1. List of code-generating scripts</td>
<td>14</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Fdmtron.c.SH</td>
<td>8</td>
</tr>
<tr>
<td>2. Two versions of fdmtron.c</td>
<td>10</td>
</tr>
<tr>
<td>3. The DRVBLD directory structure</td>
<td>12</td>
</tr>
<tr>
<td>4. The menu tree in-RAM data structure</td>
<td>18</td>
</tr>
<tr>
<td>5. Part of the menu tree data file</td>
<td>20</td>
</tr>
<tr>
<td>6. The DRVBLD top-level menu</td>
<td>21</td>
</tr>
<tr>
<td>7. A fill-in menu</td>
<td>21</td>
</tr>
<tr>
<td>8. Part of an intermediate data file</td>
<td>24</td>
</tr>
<tr>
<td>9. Overall block diagram of DRVBLD</td>
<td>29</td>
</tr>
<tr>
<td>10. Step by step use of DRVBLD</td>
<td>30</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

DRVBLD (Driver Builder) is a set of programs and shell scripts that automate the design and implementation of Device Drivers for the UNIX operating system.

The user is presented with a simple, consistent, series of menus that collect all the necessary information to automatically generate and install a device driver for the appropriate device.

Ease of operation was the main goal in designing this package. Any work that can be done without user intervention or knowledge is performed automatically. Only basic knowledge of the UNIX I/O sub-system and a detailed hardware manual for the device for which the driver is to be written are required. Ease of expansion was the second goal; thus, provisions were made for additional inclusion of data and added flexibility.
CHAPTER 2

OVERVIEW

DRVBLD consists of two main parts: the user interface, introduced briefly two sections down, and the driver generating scripts, described in the section after that. Communication between the two parts is achieved through data files that contain all the information that has been accumulated from the user and from the host system. The user interface collects data from the user that DRVBLD cannot extract automatically from system files and programs. The driver generating part makes use of this information as well as that which is gathered during the configuration phase as explained below in the next section. Additionally, some pre-packaged information is used by the code generation programs and scripts. These files supply information about common devices and protocols. The maintenance of these files is facilitated by allowing interactive additions to handle new peripherals and their methods of operation.

Configuration

Before DRVBLD can be used, it needs to be configured and installed. Two types of configuration are required.
Since DRVBLD itself has to be adaptable to the widest variety of UNIX systems in existence today, the first type of configuration insures that oddities in the host system will not cause any problems either at compile or run time. To achieve this, a "Configure" script created by Larry Wall's public domain Metaconfig package (Wall 1988), is supplied with DRVBLD. For more information on how the "Configure" script is generated and how the target package must be written in terms of the units that Metaconfig knows about as well as how custom units can be created, see appendix 1 or the documentation of the Metaconfig package in (Wall 1988). Configuring a program in this manner is simple and is a well-known procedure to people familiar with "Usenet" software.

The second type of configuration is related to the new device drivers for the host system. Much of the information can be extracted from the "include" files and other system tables. When the information is general in nature, it is collected automatically when DRVBLD is installed and processed for later use. When it is usable only by a specific type of device driver, the data are collected manually in response to user interface queries.

Adapting to the host system

This is the domain of the "Configure" script. When a
configuration script has been generated for a specific package (such as DRVBLD) by the Metaconfig program, running this script automates its installation completely. The Configure script interrogates the host system about all the relevant variations it knows about and then adapts the programs and scripts that constitute the package to that particular version of UNIX. The installer is also asked about user-configurable items. All dependencies are then extracted, a "Makefile" is generated and the package is compiled and installed. I am not the first developer to use Metaconfig to create Configure scripts. Some popular packages that use "Configure" scripts include "elm" (a mail user agent), "rn" (a news reader) and "trn" (a threaded news reader, based on rn.) However, as far as I know, I am the first to use a two-step configuration process which is in effect, a meta-meta-configuration.

Driver-related information

The configuration information described above deals with all aspects of the host system that affect the compilation and run-time environments of all user software. Other host system characteristics affect system software, such as device drivers. Information about the latter, while not needed to install or compile DRVBLD, is needed by the device drivers that it generates. Depending on whether this information is specific to a particular driver type or general in
nature, it can be extracted from the system at different times, as described below.

General information

DRVBLD scans the *.h files in the /usr/include and /usr/include/sys directories to extract necessary I/O information or system constants that all drivers need to use. This is done without user intervention. This information is extracted at installation time by the Configure script and saved in data files for later use.

Driver-specific items

Other data, such as that needed by a specific type of driver or data that is meaningful only after the user interactively selects a particular option, are obtained from the system at run time; if this is not possible, the user is queried. The information is then saved in the form of name-value pairs so it can be used as default information on a later run. This is convenient if only minor changes need to be made to a generated driver. A record of all previously generated drivers is thus kept for future use. Access to any of them is as simple as selecting a menu option or indicating the appropriate prefix on the command line. The user interface part of DRVBLD handles all this.
The User Interface

This is the only part that the user has to deal with. It is full-screen, interactive, completely menu-driven and only requires input when it is absolutely necessary. It generates the intermediate data files used by the code generator. A complete description of the user interface can be found in appendix 2.

The Driver Generator

This part of DRVBLD uses the data files created by the user interface to generate the code for the requested device driver. Once this information has been generated by the user interface, and in combination with data generated at configuration time, a second "Configure"-like process is started, this time on the code fragments that constitute the various *.SH files. These *.SH files, an integral part of the design philosophy of Metaconfig, are described in more detail below as well as in appendix 1. The complete code for the generated driver is thus put together, along with the "Makefile" to compile it and the scripts to install and test it. The installation script is particularly useful on systems that can handle dynamically loadable drivers, or to help link a new kernel.

Generating the device driver code
The data files generated by the user interface contain variables that specify what type of driver is to be generated and what its features are to be. These variables are used by the "*.SH" files. When the relevant "*.SH" files are executed, the appropriate fragments of code are written to separate files with all variable substitutions performed. Thus, the file "fdmtron.c.SH", for example, will generate the file "fdmtron.c", customized to the host system according to the information collected previously. Figure 1 shows the originating "fdmtron.c.SH" file and Figure 2 shows two output files for different configurations. In this example, fdmtron.c is the C code generated by DRVBLD for turning on the motors of the specified floppy disk drives.

Generating the Makefile and other scripts

The above procedure also generates the Makefile to compile the code derived from the *.SH files. When this is done, the dependencies in the code have to be established so that the Makefile works correctly. This is done by the "make- depend" program, which is included with DRVBLD.

Two other scripts, Install and Test, are also generated. The Install script contains the necessary commands to install the newly generated driver. This can be accomplished in a variety of ways depending on the host system. Where device drivers can be dynamically loaded, this pro-
case $CONFIG in
    if test ! -f config.sh; then
        ln ../config.sh ||
        ln ../../config.sh ||
        ln ../../../config.sh ||
        (echo "Can't find config.sh."); exit 1)
    fi
    . config.sh
    ;; esac
    : This forces SH files to create target in same directory as SH file.
    : This is so that make depend always knows where to find SH derivatives.
    case "$0" in
      */)
        cd 'expr X$0 : X')/
      esac
    esac
    echo "Extracting fdmtron.c (with variable substitutions)"
    : This section of the file will have variable substitutions done on it.
    : Move anything that needs config subs from INOSUBS! section to !GROK!THIS!
    : Protect any dollar signs and backticks that you do not want interpreted
    : by putting a backslash in front. You may delete these comments.
    : The following is included for metaconfig, since it needs explicit variable
    : names to which units to use
    $spitshefl >>/dev/null <<!GROK!THIS!
    fd_car = fdo_car = fdl_car = fd2_car = fd3_car = fd4_car = 0
    fd5_car = fd6_car = fd7_car = fd8_car = fd9_car = 0
    fd_carwo = fdo_carwo = fdl_carwo = fd2_carwo = fd3_carwo = fd4_carwo = 0
    fd5_carwo = fd6_carwo = fd7_carwo = fd8_carwo = fd9_carwo = 0
    fd0_crsav = fd1_crsav = fd2_crsav = fd3_crsav = fd4_crsav = fd5_crsav = 0
    fd6_crsav = fd7_crsav = fd8_crsav = fd9_crsav = 0
    fd_motor = fdo_motor = fdl_motor = fd2_motor = fd3_motor = fd4_motor = 0
    fd5_motor = fd6_motor = fd7_motor = fd8_motor = fd9_motor = 0
    n_of_fd = 0
!GROK!THIS!
    $spitshefl >fdmtron.c <<!GROK!THIS!
    /*
     * FDMTRON.C Turns on the floppy drive motor.
     */
    /*****************************************************************************/
    !GROK!THIS!
    if test $n_of_fd -eq 1 ; then
    $spitshefl >>fdmtron.c <<!GROK!THIS!
    void fdmtron()
    {
    !GROK!THIS!
    if test fd0_carwo="yes" ; then
    $spitshefl >>fdmtron.c <<!GROK!THIS!
    $fd0_car = $fd0_crsav |= $fd0_motor;
    }
    !GROK!THIS!
    else
Figure 1. fdmtron.c.SH.
This file, when executed, generates the driver code shown in Figure 2.
(Continues on next page.)
void fdmtron(fd_num)
{
    switch (fd_num) {
        case 0:
            while test $i -lt $n_of_fd
                do
                    eval csrwo=\d\$i\_csrwo
                    if test "\$csrwo" = "yes" ; then
                        eval cntrl_reg=\d\$i\_cntrl_reg
                        eval cntrl_sav=\d\$i\_cntrl_sav
                        eval motor_bit=\d\$i\_motor
                        case $i :
                            $cntrl_reg = $cntrl_sav |= $motor_bit;
                            break;
                        !GROKITTHIS!
                    else
                        eval cntrl_reg=\d\$i\_cntrl_reg
                        eval motor_bit=\d\$i\_motor
                        case $i :
                            $cntrl_reg |= $motor_bit;
                            break;
                        !GROKITTHIS!
                    fi
                    i='expr $i + 1'
                done
        default: panic("Invalid Floppy drive number");
    }
}

: In the following dollars and backticks do not need the extra backslash.

$spitshell >> fdmtron.c <<<!GROKITTHIS!

$eunicefix fdmtron.c
chmod 755 fdmtron.c
$eunicefix fdmtron.c
FDMTRON.C Turns on the floppy drive motor.

void fdmtron()
{
    *DISK_CNTRL = dcr_save | FDMTRON;
}

Figure 2a. Routine to turn a floppy drive motor on. This one was generated for a driver that controls only one drive. Note that symbolic constants were found in the system files and so are used.

void fdmtron(fd_num)
short fd_num;
{
    switch (fd_num) {
    case 0:
        *((ushort *)0x4E0000) = fd0_cr_sav |= (ushort)0x0008;
        break;
    case 1:
        *((ushort *)0x4EF000) |= (ushort)0x0010;
        break;
    case 2:
        *((ushort *)0x4EE000) = fd2_cr_sav |= (ushort)0x0008;
        break;
    default: panic("Invalid Floppy drive number");
    }
}

Figure 2b. Same as above, except that this routine is for a driver that controls three floppy drives. No symbolic constants are used here. The control registers for floppy drives 0 and 2 are write-only, that of drive 1 is read/write.
procedure is used; otherwise, standard commands to link a new kernel with the new driver are generated. The Test script contains a series of commands that read, write and perform other appropriate operations through the newly created driver. The appropriate test programs are also generated.

The DRVBLD directory tree

All support programs and data files are kept in a separate directory tree structure. The path to the root of this tree is specified at configuration time as an answer to a question asked by the configuration script. An environment variable, if it exists, can override this path. Under the DRVBLD home directory, three sub-trees contain all the DRVBLD files, these are:

1 - $DRVBLDHOME/drivers
2 - $DRVBLDHOME/code
3 - $DRVBLDHOME/data

Figure 3 represents graphically the DRVBLD directory tree. Its content is described in detail in the section of the same name below.

*.SH files

These are the shell scripts that when executed, generate the actual C code that will make up the new driver, as well as the "Makefile" and auxiliary scripts. Every SH file generates one C routine or script, with the appropriate
Figure 3. The DRVBLD directory structure.
substitutions. See Figure 1 for an example SH file and Figure 2 for the code it generated.

The names of the *.c routines generated reflect the name of the driver, thus bk/sh is replaced by the appropriate prefix, which has already been checked for conflict with existing drivers on the host system. For example, Table 1 lists the generic routines for block and character device drivers, in addition to some others. When DRVBLD generates a device driver, it replaces the generic prefixes "bk" and "sh" by the one specified by the user, say "sd".

Other routines might be added later to handle different versions of UNIX that might need them. Table 1 lists some of these driver-specific local routines and some generic functions and scripts.

In order to avoid duplication of code, some device drivers control more than one device of a similar nature. Only low-level code is different for the different devices, which usually appear to the operating system with the same major number and different minor numbers. Device drivers that handle multiple devices at the bottom end using a common top end are described in (Drake 1991). This type of driver design was also used in the "gd" driver of the AT&T 3B1 (AT&T 1985). The current revision of this document, which can be found in various Internet archives is the
Table 1. List of code generating scripts.
authoritative guide for writing device drives for that machine.
In a similar way to device drivers themselves, DRVBLD consists of a top half that deals with the user and a bottom half that generates the code. The two types of UNIX device drivers, Block and Character (Pajari 1987), have been divided according to the functions they perform. These functions are consistent throughout all versions of UNIX. The task of generating device drivers was further subdivided into smaller and smaller tasks. At various points, the differences between implementations of UNIX appeared. These differences were then incorporated into the "intelligence" of DRVBLD in the form or Metaconfig Units. Since each Metaconfig Unit handles one incompatibility, or at most a few, making DRVBLD more capable only involves adding more Metaconfig Units. Metaconfig Units (also referred to as *.U files) are special-purpose files. Metaconfig handles all the details such as dependencies of variables or units, or the order in which they are dealt with. The user interface of DRVBLD replaces the interactive part of the standard Metaconfig-generated "Configure" script during the driver generation session. Expanding DRVBLD only involves adding
new menus, new Metaconfig units and *.SH files. This is all that is needed to accommodate new devices, new versions of UNIX, new hardware, new protocols, or just new features or optimizations. Once these additions have been made known to DRVBLD, they remain available for the future generation of all device drivers that might use them. For instance, if a slightly more efficient way is found to load buffers using DMA on a particular version of UNIX, the improvement can be saved so that all subsequently generated drivers can use it.

Since improvement and optimization follow essentially the same path as in normally generated software, there is no reason for DRVBLD generated device drivers to be less efficient than a similarly refined driver written manually. The *.SH files can contain any type of code, even assembly language fragments for those regions where timing and speed are critical. DRVBLD can very easily know what code to use for what processor or family of processors from the extracted information.

The following two subsections describe the user interface followed by the driver generator code.

The User Interface

At the heart of the user interface is the menu tree (see Figure 4.) Upon start-up, this tree, which resides in a plain ASCII file, is read in. Part of the information that
Figure 4. The menu tree in-RAM data structure (one node.)
describes each node allows the program to build the in-memory representation of the menu tree with the proper topology. This tree can be completely arbitrary and there are no built-in limits of any kind; growth can occur as long as memory is available.

The Help file is read next and its information, properly broken up, is then linked to the RAM-resident menu tree. Figure 4 illustrates the organization of the menu tree data structure along with the associated help text. Figure 5 shows the first few nodes in the menu tree, as they are saved in a file.

After the menu tree, the help text, and configuration information have all been read in, what the user first sees depends on a number of factors. If the program has never been run before, the introductory text is displayed. If no command line options indicate otherwise, the top level menu is displayed. Figure 6 shows the top level menu. Figure 7 shows a partially filled fill-in menu, which is the type used when specific user input is required. Other items depend on default or previously configured options saved in the configuration file.

Every time the user makes a decision, that choice is recorded as the value of a variable. The resulting set of variables eventually becomes the output of the user inter-
Figure 5. Part of the menu tree data structure, as it is stored on disk.
Device Driver Generator - Main Menu

1 - TTY - character device
2 - DISK - random-access block device
3 - TAPE - sequential block device
4 - LAN - bi-directional character (packet) device
5 - OTHER - new type of device
6 - Exit Driver Building Utility

Select only one type of device to build the driver for "?" or "h" for help

Figure 6. The DRVBLD top-level menu.

Floppy 0 control register

- Control register address: 0x4E0000
- Is this address to be accessed as 8, 16 or 32 bits? 16
- Is the control register write only? Y
- Internal variable to hold its status: fd0_cr_sav

Hit return to go to the bit definitions menu

Figure 7. A Fill-in menu. User input appears underlined.
face. This output is then used, as shell variables, by the code generation part of DRVBLD. Figure 8 shows part of this information.

The Driver Generator

Files used and generated

The extensible nature of DRVBLD requires standard places for both permanent and new data. Upon installation, a directory tree structure is created that consists of three sub-trees:

1 - $DRVBLDHOME/drivers
2 - $DRVBLDHOME/code
3 - $DRVBLDHOME/data

Figure 3 shows a graphical representation of the directory tree.

If no override has been specified through the environment variable $DRVBLDHOME, the standard location, set at configuration time, is used. Since generation of code requires write access to the $DRVBLDHOME/drivers directory, and user-implemented extensions are saved in $DRVBLDHOME/data, users must belong to an authorized group, which has write permission on those directories. An alternative is, of course, to specify a $DRVBLDHOME directory that the user owns. In such a case, to avoid having to duplicate all the code in $DRVBLDHOME/code, the original
directory is used if the necessary files are not present in the user's own directory. This also allows the user to use modified files, which are then used instead of the standard ones.

Drivers

This directory is originally empty. It is used to save a record of the choices the user makes in the process of generating a particular driver. A sub-directory is created for each such driver whose name is the prefix for that driver. This information is recorded in plain ASCII files in the format described in chapter 4 and as shown in Figure 8. Each previously generated driver can be easily modified or re-generated through the use of these files. General information that is independent of any of the drivers being generated, or is not directly related to device drivers but necessary for the process of generating them, is kept in this directory proper.

Code

The two directories in this sub-tree, $DRVBLDHOME/code/bk and $DRVBLDHOME/code/ch, contain all the routines used by DRVBLD to generate drivers for block devices and character devices respectively. Table 1 lists the contents of these two directories. This code takes the form of *.SH files or regular C functions and includes the global
Figure 8. Part of an intermediate data file. Information is passed from the user interface to the driver generator scripts through these shell variables.
driver routines, local routines and miscellaneous code. The *.SH files are executed in the generation process and use both the output of the user interface and the relevant data files from the third sub-directory, described below.

**Data**

This directory will contain additional data and code. Two sub-directories, $DRVBLDHOME/data/hardware and $DRVBLDHOME/data/protocols, will consist of hardware and protocol descriptions respectively. This directory is presently empty.

As planned, each file in the hardware directory will be the description of a particular piece of I/O hardware, such as a specific UART chip, a controller chip, or a whole I/O sub-system board, such as a disk controller or a tape controller. The format in which this information is to be kept will be defined in the next phase of this project and will, of course, be software oriented. Nothing in this directory is, strictly speaking, necessary run DRVBLD. However, it will be much easier to specify a certain piece of hardware rather than all the logic details of such item, such as bit definitions for a particular register, as well as their characteristics (how wide, read/write, read only, write only, etc...) Following the general extensibility scheme used throughout the design of DRVBLD, addition of this
information can be interactive. In such a case, the user interface will guide the user in the creation of the appropriate new menus, Metaconfig Units and SH files. The resulting information and code will then be available for later use.

The "protocols" directory will contain protocol descriptions. Here again, each file will consist of a description of a specific protocol. As above, these are not strictly necessary, but they will make it much easier to specify, for example a "SCSI driver", rather than have to enter all the details of the SCSI protocol. These files will be code fragments, which would be included in the generated drivers, that embody the logic necessary to communicate with devices using the specified protocol. Some systems also provide generic code that can be used without change to generate drivers for a wide variety of devices. Such is the case with the SCSI drivers discussed in (SCO 1989). DRVBLD will make use of such code whenever possible.

How the code is generated

The final product of this package is a complete device driver tailored to the hardware it's supposed to work with. The input to this process comes from three sources:

1 - A scan of /usr/include/*.h and /usr/include/sys/*.h
2 - Information gathered at DRVBLD installation time
Scan of *.h files

The most straightforward use of header files, as it relates to device drivers, is of course their inclusion in the preamble. Not all drivers need the same *.h files and not all systems supply all of them. One of DRVBLD's tasks is to determine what is available and what is needed.

A more complicated task is to search for system constants, addresses, and structure definitions. These might not be in the same files on all systems or might not even have the same names. When they are found, their values are saved. Later in the process, when the *.SH files are executed, the generic shell variables contained therein are replaced with the system-specific values, and thus correct C code is generated.

Information gathered at installation time

Similarities exist between the way DRVBLD itself is configured, compiled and run, and the configuration of the device drivers it is supposed to generate. It is thus no surprise that some of the information collected from the host system in order to compile and install DRVBLD is also usable by the device drivers generated. This information is saved in $DRVBLDHOME/drivers when DRVBLD is first configured
to be used when needed. Figures 9 and 10 show how this one-time operation fits in with other DRVBLD tasks. Figure 9 shows the overall structure of this package. Figure 10 breaks down tasks chronologically. Saving this information when first extracted shifts the time this operation takes to the original installation and so avoids repeating it for every driver generated.

**User input**

When it is impossible to obtain the required information from the system, as in the case of the features of the hardware for which the device driver is to be written, the user must be queried. For example, if a particular I/O card has control and data registers at addresses X and Y (these being possibly configurable), DRVBLD needs to know them in order to generate a driver that will properly talk to the hardware. After the user has entered them (in hex, octal or decimal following the C conventions), the system header files are scanned. If the address just entered is found, the user is notified of the possible conflict and asked to confirm the previous selection. If the address is in fact correct and was included in the system files for future expansion, DRVBLD will use the symbolic constant defined; otherwise, it will use the straight address. This datum then becomes part of the output of the user interface which takes the form of shell variables and their associated
Figure 9. Overall block diagram of DRVBLD.
Configure and install DRVBLD.  

Directories are created, data installed in proper places. DRVBLD executables placed in $PATH.  

Run DRVBLD (interactive session with user).  

Driver configuration data is collected.  

User ends interactive session and requests that DRVBLD generate a driver.  

Driver code is generated as well as the Makefile, Install script and Test script.  

Compilation, installation, testing of generated driver.  

System with new device driver.  

**Figure 10.** Step by step use of DRVBLD.
values.

When more than one match results from this scan, such as could be the case with register bit masks, the user is presented with all occurrences found and allowed to choose one or, if none is appropriate, to define a new constant.

**Executing scripts and generating code**

All the information collected in the steps above is now ready to be used. The environment inheritance feature of the UNIX shell makes it automatic to substitute all the local values as the *.SH files are executed.

When the user has finished entering the requested information, the user interface part of DRVBLD writes out all the collected information to a file and then the master script is executed. This script in turn executes the appropriate *.SH files, which in turn output the device driver C code, along with the necessary Makefile, installation and test scripts, as well as other test programs.
CHAPTER 4

SIMPLE EXAMPLES

Two simple but complete pseudo-device drivers were generated: a RAM disk driver and a pseudo-tty driver, based on the ones described in Egan and Teixeira (Egan, Teixeira 1988.) They were configured for three systems: an AT&T 3B1 with AT&T UNIX System V release II, a Harris 386 running UNIX System V release 3.2 and a generic 486 running SCO UNIX System V. As of this writing, final adjustments are in progress for other hardware and versions of UNIX, including 68020/BSD 4.2, micro-vax/BSD 4.3, NeXT/Mach, and RS6000/Aix. Since these drivers are pretty generic, the configuration differences are not very extensive, but because of DRVBLD, they have been handled automatically. More complicated drivers for real devices only require incremental additions. As was shown earlier, Figures 1 and 2 show just one such addition: the *.SH file and resulting code to turn a floppy drive motor on, for a couple of different configurations: a driver that supports one floppy drive only and one that supports three. In the first case, symbolic constants were found that represented the address and bit mask specified, so they were used.
What the user sees

Figure 6 shows the first screen displayed by DRVBLD. A single keystroke takes the user to a sub-menu with the same format. This process is repeated as many times as necessary to narrow down the user's intentions enough to start asking for system information. The second type of menu, the fill-in menu, is then displayed. Figure 7 shows one fill-in menu. User input is the underlined text. The menu following this one is a multiple-choice menu where the bits of the control register are defined. While the session is going on, DRVBLD is assigning values to all the variables it knows about. Upon completion, and after user confirmation, the driver generator starts its work, using as its input the intermediate data files generated by the user interface. Part of one such file is shown in figure 8. This intermediate data takes the form of shell variable assignments. These are the shell variables that will be replaced by their values in the *.SH files. The comments supplied are automatically inserted by the user interface for ease of debugging and expansion, since DRVBLD allows interactive addition of menus for new types of drivers.

What the system generates

Figure 1 shows a very short *.SH file which nonetheless exemplifies both the substitution of variables and the gen-
eration of different chunks of code according to the desired local configuration. Figure 2 displays the resulting code generated for a couple of configurations. From the user responses shown in Figure 7 and what DRVBLD was able to gather from the system, the appropriate values were inserted in the intermediate variables as shown in Figure 8. All the necessary details to generate correct code are there. When the SH file shown in Figure 1 is executed and the variables substituted, the code is generated. Of course, this floppy motor code is but a small part of a real device driver, but it shows the procedure used to generate code with DRVBLD. It is apparent that this process shifts the complexity of dealing with multiple machine architectures or types of devices to the configuration process. Much of this is done without user intervention and when user input is required, such input is restricted and checked for correctness. Help is also given interactively.

A Makefile, an installation script and a test script are also generated. Of course, installation and testing can only be done with root access, even in systems that allow dynamically loaded drivers, but the procedure is greatly simplified nonetheless.
CHAPTER 5

SUMMARY & CONCLUSION

The main goals in the design of DRVBLD, a device driver generator, were simplicity of use, expandability and portability. The menu-driven user interface with on-line, context-sensitive help and a consistent set of commands insure that the user cannot get lost. New devices are easily accommodated by adding new menus, new Metaconfig units, and new *.SH files that can be derived from existing ones. While the process of expanding DRVBLD by adding new types of devices or versions of UNIX is more involved than just generating a driver, it is still easier than writing a driver from scratch.

DRVBLD has been designed with modularity in mind, so adding support for new types of devices, new protocols, or new hardware as they get developed is as simple as adding new *.SH files or new hardware or protocol definition files. I envision DRVBLD becoming more capable of generating more complex drivers as the modular additions for which it was designed are implemented. At some point in the future it will be able to generate robust device drivers for real-life
devices.
APPENDIX A

INTRODUCTION TO METACONFIG
Introduction to Metaconfig

Metaconfig, written by Larry Wall, is a perl program that, along with a few other auxiliary scripts was designed to simplify writing portable programs. It is part of the "dist" package, which also contains programs to generate patches for packages that use RCS, and makedist which generates "shar" kits.

Metaconfig generates a "Configure" script that is distributed with the target program. The end-user runs the "Configure" script, which is a regular shell script, and then compiles the program. Makefiles distributed with "Usenet" software usually have a "make install" option, so typing "make" (to compile the code) and "make install" (to install it, usually as root) is all the user has to do before actually using the program.

The "Configure" script, when executed, checks a few things about the system and asks the user about configurable items, such as where to put the manual page, where the executable file should go, program defaults, etc... It also can define or undefine constants to include or exclude blocks of code and substitute variables in SH files, which are just shell scripts. The "Configure" script makes it
possible to install and run a program written with portability in mind on just about any version of UNIX in existence.

The "Configure" script gets its code from "Metaconfig Units" which embody what metaconfig knows about portability. Units (or *.U) files are a special type of script. Certain lines, that begin with meaningful (to metaconfig) keywords, are used for its own purpose. The rest are just shell code that is included in the "Configure" script. Metaconfig also knows about the files that constitute the target package, which are listed in the "MANIFEST" and executes the scripts, performing the variable substitutions.

A standard set of *.U files can handle just about every variation in past and present UNIX systems; however, new *.U files can be created, giving Metaconfig an extensibility that was heavily used in the design of DRVBLD. Under normal circumstances, in the one-step configuration procedure for which metaconfig was designed, additional Units are rarely needed.

The current production version of Metaconfig is 2.0, and has been since 1988. Version 3.0 is about to be released "any day now". Metaconfig 2.0 was written in the days of Perl 2.0. The current version of Perl is 4.0. Perl 3.0 was not totally compatible with Perl 2.0, so Metaconfig 2.0 had to be modified slightly to make it work.
APPENDIX B

DRVBLD MANUAL PAGES
NAME
drvbld - Device driver builder

SYNOPSIS
drvbld [-=] [-t type] [-d device] [-p protocol] [-r] [-s]
[-v] [driver_prefix]

DESCRIPTION
Drvbl (Driver Builder) is a set of programs and shell
scripts that automate the design and implementation of Dev-
ice Drivers for the UNIX operating system. Only basic
knowledge of the UNIX I/O sub-system and a detailed hardware
manual of the device for which the driver is to be written
are required.

Drvbl is full-screen, interactive, completely menu-driven
and only requires input when absolutely necessary. All
prompts and displays are consistent and the commands have
identical behavior at all levels.

Beginner and Expert modes are supported. In Beginner mode,
the user receives a brief introduction before each menu or
prompt and the help screens are more detailed than those
available in Expert mode. In the latter, no supporting text
is provided and help screens provide more concise informa-
tion. The Expert mode is temporarily over-ridden if the
user requests help twice in a row at any point. In such a
case, the more complete Beginner help is displayed at the
second request, for that request only.

On-line context-sensitive help is available at all times.
Two levels, selectable by a user-configurable toggle switch
provide basic and lengthy information, suitable for
beginners, or minimal descriptions for experienced users
that only need to be reminded briefly of the available
options.

All user input that is easily reversible, such as menu
selections, is done in cbreak mode (ie: without waiting for
<RETURN>). Text input is checked for correctness as much as
it is possible to do so before allowing the user to proceed.

Command line options are available to either specify over-
rides to configured options or to skip any number of menus
and directly start the program at the particular level of
interest. This is a useful short-cut for experienced users.
Specifying a device driver prefix on the command line that
is known to drivbld starts at the appropriate menu, with all
the defaults set to what was input the last time that par-
ticular driver was modified.
Upon startup, `drvbd` does several things:

1. It checks the environment for applicable variables.

2. It looks for the user’s configuration file (`.drvbdrc`) normally found in the user’s home directory, but its location can be over-ridden with an environment variable. If this file does not exist, `drvbd` assumes that it has never run and displays an introductory message this time only. Then a configuration file with default parameters is created. The configuration parameters can be changed through the configuration menu.

3. It inputs the user’s `.drvbdrc` file (if it was not just created with default values.)

4. It reads the menu file and the help file.

5. It displays the appropriate initial menu screen. Which one this is depends on the command line options and the driver prefix specified. If `drvbd` is invoked with no command line arguments, the first menu is displayed. Command line options or a driver prefix that is known to `drvbd` will skip to the appropriate menu. The amount of detail displayed reflects the mode that is currently in effect.

Menus

Two types of menus are supported at this time. The conventional type (in which the user can select one item out of the few presented) is most useful in the beginning stages, when the user is deciding what type of driver to create. When more specific information is required, the user is asked to enter data in response to individual questions. Both multiple choice and fill-in menus recognize common commands, such as requests for help, default actions, cursor movement and item selection.

`Drvbd` does not require separate configuration to add new menus, as would be required, for instance, to create a new type of driver. All menus have one option that returns to the previous level and one that allows for the creation of a new menu. The procedure to add a new menu is documented in detail in the on-line help.

Valid input

Three types of user commands are recognized to:

1. Request help
2. Move around the different options

3. select an option or enter a value

At any time, pressing the '?' key brings up a window and the appropriate help text is displayed, depending on the current menu and help level. A second help request displays the more complete beginner text if the help level was expert, this time only. The pager used to display help messages accepts its own commands, which are a subset of the standard UNIX text display programs. Standard UNIX features are supported. For instance, a shell can be obtained at any time by typing '!'. Any command can thus be executed, or a sub-shell run by typing the name of the desired shell after the '!'.

Arrow keys, when supported, always move to the previous/next option. Other functions, such as selecting an option, taking the default path or entering numbers or text, require different input in multiple choice and fill-in menus. The expected input is always explained in the prompt line at the bottom of the screen. To select a particular option, the RETURN key is pressed. We have attempted to keep all the commands as consistent as possible throughout the user interface; however, in some situations where a different type of user input is being requested, the same input has to be interpreted in different ways. The first obvious example is the two types of menus supported.

Multiple choice menus

The user input that functions in a unique fashion in multiple choice menus is as follows:

Numbers: A number will select that particular menu entry identified by the number entered; if the number is out of range, the appropriate error message is displayed in the error message line and the bell sounds.

Space: The space bar always takes the default action. In this case the default action is to go to the next entry or wrap around to the first one if the last one was currently selected.

Return: The return key always selects the currently highlighted option. The resulting action is to go the next/previous menu, as indicated in the selected option.

Fill-in Menus

Fill-in menus are used when specific user input is required. Such input can be numeric (in decimal, octal or hex),
alpha-numeric, or YES/NO answers. Individual menu items might have a default value (selected when the user just enters "return") but otherwise user input cannot be avoided.

Since numbers can be valid input in response to certain fill-in questions, fill-in menu items are not prefixed with numbers, like in multiple-choice menus. Each fill-in menu datum entered is checked to see if it is of the type requested. Fill-in items are accepted when the "return" key is pressed. If a default exists, either the space bar or the return key with no previous input selects it.

The help pager

The pager supports a subset of common UNIX pager commands. Arrow keys, if supported, work in the predictable fashion of scrolling up or down one line. The space bar takes the default action of displaying the next screen, as does ^F (like in vi). ^B goes back one screen and 'k' and 'j' scroll up or down one line, again as in vi.

OPTIONS
-= The current invocation is to use the same options and parameters as the last one.
-t type
    The type of device driver to be generated. Type can be disk, tape, tty, net or possibly other types to be added later. The characteristics that define each type are saved in a data file.
-d device
    The (hardware) device to generate this driver for. The characteristics of each device are saved in a data file. The amount of detail varies with each device. The information required for, say, an RS-232 UART will necessarily be less than that for a full board with multiple I/O ports.
-p protocol
    The protocol to be supported by the driver. This can be a communication protocol from serial lines or network drivers, or a mass-storage interface definition. Examples of protocols include UUCP, SCSI, IP, etc...
    This information is also saved in a data file.
-r Generate code for the raw device too. This, among other things, will generate a driver installation script that will create the raw device entry in /dev.
-s Start execution with the setup screen. In this way, the user can set up the various options to individual
taste. This information is saved in the user configuration file.

-v  Display the version and patch level, then exit.

ENVIRONMENT
Drvbld looks at environment variables on startup so that its behavior may be altered without re-compiling or changing the configuration file. The following variables are known at this time:

DRVBLDRC
contains the path to the configuration file (.drvbldrc) whose default location is the user’s home directory.

DRVBLDHOME
contains the path to the root of the drvbld directory tree. A default value is specified at configuration time but an override with this variable might be useful for testing.

FILES
$DRVBLDHOME/help.txt  On-line help text
$DRVBLDHOME/drivers/xx/*.sh  Config. files for driver xx
$DRVBLDHOME/drivers/xx/*.U  Metaconfig units for driver xx
$DRVBLDHOME/code/bk/*.SH  Code for block drivers
$DRVBLDHOME/code/ch/*.SH  Code for character drivers
$DRVBLDHOME/data/hardware/*  Hardware description files
$DRVBLDHOME/data/protocols/*  Protocol description files

DIAGNOSTICS
Drvbld

All user errors are reported interactively. Additional information can be obtained using the on-line help facility.

Generated drivers

A set of macros and debugging variables can optionally be included in the generated drivers to assist in troubleshooting.

LIMITATIONS
Only simple drivers can be generated at this time.
APPENDIX C

BRIEF INTRODUCTION TO UNIX DEVICE DRIVERS
Brief Introduction to Unix Device Drivers

According to Egan and Teixeira (1988), "A device driver is a collection of subroutines and data within the UNIX kernel that constitutes the software interface to an I/O device. When the UNIX kernel recognizes that a particular action is required, it calls the appropriate driver routine. No other code within the kernel makes direct contact with the device. It is possible (but rare) for a user process to map device registers into its virtual address space, but only a device driver can respond to device interrupts."

Table 1 lists all the common global driver routines, as well as some possible internal ones. Of course, not all drivers need to use all these routines, a printer driver, for instance, needs no read function. A UNIX device driver consists of a top half, that deals with the user's process and a bottom half, which deals with the device it controls. Some of the global driver routines (or entry points) run in response to some request generated by the user process, and some are run by the kernel or the hardware. For instance, when a user program needs to read from or write to a device, the kernel runs the read or write routine of the appropriate
device driver on behalf of the user process. This routine initiates the transfer and then goes to sleep to allow other processes to execute, since I/O transfers are usually very slow in terms of CPU speed. When the transfer is complete, the device signals this fact through an interrupt. The interrupt service routine of the driver then handles the completion of the operation.

Since UNIX is multi-user and multi-tasking, access to the device a particular driver controls must not be allowed freely. If multiple processes require access to the device simultaneously (simultaneously here meaning while an I/O operation is occurring on behalf of one process) the driver has to organize access to the device in an orderly fashion so that parallel executions do not interfere with each other.
REFERENCE WORKS


The Santa Cruz Operation. 1989. SCO UNIX system V/386 development system device driver writer's guide.