Setting up a Portable Intellivision Development Environment on Your Android Device

Written by Michael Hayes

intylab@yahoo.com

Date of Last Modification: May 31, 2019

(Note: don't let the length of this document intimidate you. It's designed to be easy-to-follow, not concise. Also, this is something you will only have to do once per device.)

Introduction

You have a portable Android device (phone or tablet) and a physical keyboard connected.

You now have some experience developing in IntyBASIC.

You would like to do future development using only your device and keyboard, so you can develop anywhere you're at in the cracks of time in your busy schedule.

You may not know a darn thing about Linux and can't be bothered to "root" your device.

This document is for you.

Disclaimer

I feel it is imperative to put this on the first page:

Neither I nor Midnight Blue International, LLC are responsible for anything bad that happens to you or your device for the use of any of the information in this text.

Neither I nor Midnight Blue International, LLC are responsible if you get fired from your job because you got caught writing games on company time.

Neither I nor Midnight Blue International, LLC are responsible if your Life Partner walks out on you because you're too busy making games anymore.

Standard data rates apply with your mobile carrier, blah blah blah.

You will need:

- A device with Android 5.0 or higher and about 600M internal storage space.
- A physical keyboard for your device.
- I recommend a mouse for your device as well, but it's not required.

Table of Contents

Revision List

May 31, 2019

- I created this new section for Revisions.
- I made a correction to the part about taking full-screen screenshots. The parameter for "import" is "-screen", not "-session". I also fixed a typo in the sentence about references to "termux-storage-get".
- I expanded the "Outside World" notes to include connections to External Storage as well as Internal Storage.
- I added a note about "hackfile.cfg" to toggle between standard controls and an emulated ECS keyboard.
- I added a section to "Additional Steps" so you can define the default settings for Aterm and not need all those command-line parameters.
- I created an Appendix to create a script for playing games in this environment.
- I changed the illustration of the screenshot from my tablet to demonstrate that I am now running wm2 as my Window Manager. I also added a blurb about wm2 in the Musings.

Additional Notes

- Words that are underlined have a Glossary entry. Use the Glossary if you're having trouble understanding the context. Use of words you're probably not familiar with is unavoidable.
- Whenever there is text to enter, either within a text file or as a command, … `it will look like this.`
  Be sure to enter it exactly as it appears, including capitalization. Pay careful attention to spacing as well.
  - If there is text that spans multiple lines, I put spaces between lines so you know where the line breaks are.
  - If a single line of text is too long to fit into a line within this document, it wraps around onto the next line without a space.
  - Sometimes, you will have to make substitutions appropriate for your project or environment. Where there will be substitutions, I color the text in amber … `like this`

Step 1: Install some apps onto your device.

You should at least be familiar enough with Android to be able to install apps onto your device, all the more so if you could figure out IntyBASIC development.

Option 1 (recommended): Using F-Droid

A. Go to https://www.f-droid.org and click on the "Download F-Droid" button.
B. When finished downloading, you should be prompted to install it. If not, find the .apk file in your Downloads folder and run it manually.
C. You will probably get a warning about trying to install an app from an unknown source. Make the necessary changes to allow it to install.
D. When it is finished installing, you should be prompted to Open / Launch / Run it. Do that now.
E. Tap on the Search button (a magnifying glass symbol within a round green button in the bottom right corner, just above the selection bar across the bottom) and type "Termux".
F. Follow the prompts to install Termux. Don't worry about launching it just yet.
G. Search for "XServer" in the same manner as before and install that.

Option 2: via Google Play Store

A. Go to the Google Play Store and install the following apps:
    I. **Termux**: A Linux environment for Android that works without rooting.
    II. **XServer XSDL**: A graphics server that works with Termux to provide a graphical interface as well as sound.

Step 2: Install the text editor of your choice.

Before opening Termux, first you can take a moment to "shop around" for a text editor of your choice that you will use within the Termux environment. This is a decision I am leaving up to you.

I had to choose one myself, and *emacs* is what I chose, since I have been using it since my college years. It's about 100M in size though, and even more if you install the graphical components later. Plus you'll need to spend some time with the built-in tutorial until you become familiar with the keyboard shortcuts, if you're working in the text-based environment.

If storage space is an issue for your device, than I can recommend *micro*. It's only 10M in size. There's also *vi*, which is pre-installed on Termux, so if you're familiar with it and don't want to install something else, then you can skip ahead to the next step.

A. Open your web browser.
B. Go to https://wiki.termux.com/. This is the official Termux Wiki page.
C. Scroll down to "Software" and select that.
D. Select "Editors".
E. Select "Text Editors".
F. Decide on a text editor you like best.
G. Most of the available text editors have instructions on how to install them within Termux. Open Termux now and enter the command. For example, to install emacs, enter the command
```
pkg install emacs
```
H. You'll be prompted during the installation how much additional space will be allocated. Enter "y" to confirm.

Step 3: Enable Android API calls.

This is necessary for future steps.

If you don't know what "API" means, take a moment to look it up if you prefer. Bear in mind though, the purpose of this manual is just to get you up and running doing Intellivision development on an Android device, not to give you a full lecture about Linux.

A. Go back to the Google Play Store or F-Droid and install Termux:API
    - Note: you need to install Termux:API from the same source that you installed Termux from.
B. Go into Termux again (the session continues running until you tell it to stop) and enter the command
   ```
   pkg install termux-api
   ```

Step 4: Establish contact with the outside world.

Because we're creating a Linux environment without rooting, there are a few workarounds we'll have to implement so we can import files into the "Linux filesystem" and also export files to the Android filesystem.

A. Enter the command
   `termux-setup-storage`
B. Your device will probably ask you to grant Storage permission to Termux at this point. If you're using Version 5 of Android, it might not ask.

So what did this do?

If you enter the command "ls", you will notice there is now a folder called "storage" which didn't exist before. (You know it's a folder because Termux colors it violet by default.) Within this folder are multiple symbolic links, which Termux colors cyan.

The one link called "shared" points to your device's Internal Storage. The link called "external-1" points to your device's External Storage, starting at path "/Android/data/com.termux/files".

From the Linux filesystem, you can now export files to your device's /sdcard folder via /storage/shared. You can also update and delete files out there.

What you probably can't do is import files, not directly anyhow. That's where the next steps come into play.

Step 5: Download IntyBASIC.

We are going to get the latest version of IntyBASIC (v1.4.0 at this time of writing) right from the source, and put it directly into our Linux filesystem.

A. Enter these commands:

```
pkg update && pkg upgrade
pkg install git
pkg install proot
pkg install wget
git clone https://github.com/nanochess/IntyBASIC
```

You now have a folder called "IntyBASIC" with IntyBASIC in it.

In case you don't know, file and folder names in Linux are case-sensitive. "IntyBASIC" and "intybasic" are not the same, for example, and both can exist within the same folder.

Step 6: Install IntyBASIC.

For those of you who are new to Linux, get used to the idea of having to compile the source code for software packages and not having pre-compiled binaries (executable files). If a software package is available as an installable package (such as the text editor from earlier), count it a blessing.

Fortunately, I created this document to hand-hold you through this process.

A. Enter these commands:

```
cp IntyBASIC/intybasic/*.asm .
cp IntyBASIC/intybasic/*.cpp .
cp IntyBASIC/intybasic/*.h .
pkg install clang
clang++ IntyBASIC.cpp code.cpp microcode.cpp node.cpp -o ./intybasic_termux
```

- Note: the fifth command here is a single line, as I mentioned earlier. It doesn't fit into a single line in this document. Make sure there's a space between "node.cpp" and "-o".

You might get a warning during compilation here, but it should successfully create the executable file "intybasic_termux", which is what you'll be using.

Step 7: Download and Install jzintv/as1600.

We will use a different method than before to get it into the Linux filesystem. Also, because jzintv is the actual emulator and provides graphics and sound, you will have to install it yourself, as you did with IntyBASIC.

A. First, enter the command
```
uname -m
```
This will display your device's architecture.

B. Open your device's web browser and go here:
atariage.com/forums/topic/283347-portable-intybasic-development-environment

C. Scroll toward the bottom and find a download of jzintv that matches your architecture. If there are no matches, then download either one.

D. Go back to Termux and enter the command
```
termux-storage-get ./jzintv.zip
```

E. Android should now pop up a File Selector window. Go to your Downloads folder and select the file you downloaded.
   - There's a chance the Android File Selector will not appear, if you're using an older version of Android. If that is the case, you can substitute the command "cp storage/downloads/jzintv-20181014-1791-arm.zip ." and at the next step, enter the command "unzip ./jzintv-20181014-1791-arm.zip"
   - From this point forward, any references to "termux-storage-get" you will have to substitute with the command to copy the file directly from the appropriate folder.

F. Enter these commands
```
unzip ./jzintv.zip
pkg install make
pkg install sdl
pkg install sdl-dev
cd jzintv-20181014-1791/src
make -f Makefile.termux
```
This is the first big moment of Truth: does jzintv compile? I had a *lot* of help from Joe Zbiciak to get it working in Termux! First, I got it working on my phone (aarch64), and then with some changes, I got it working on an old tablet (arm), which is the "worst-case scenario" device that can run Termux at all.

G. Finally, with jzintv compiled, enter these commands
```
cd
cp jzintv-20181014-1791/bin/as1600 .
cp jzintv-20181014-1791/bin/jzintv .
```

Step 8: Get the rest of the files you need.

Among those files are exec.bin, grom.bin, and ecs.bin, as you probably already know. You should know how to get them onto your device, along with any source files from your current project if you are working on one.

A. Enter the command
```
termux-storage-get ./exec.bin
```
Once again, you will be taken to the File Selector. Select the exec.bin file from wherever you have it.

B. Enter the command
```
termux-storage-get ./grom.bin
```
Select the grom.bin as before. Note we are naming the imported files in all lower-case. Remember, file names are case-sensitive in this environment.

C. If you have ecs.bin and want to import that as well, you may do so.

D. If you have any existing projects you want to import, then get out of Termux for a moment and perform the following steps:
   I. Using your device's built-in Storage app (probably called "My Files", but it varies across devices), create a folder called "Archive" in your device storage. (This is where we'll back up project files outside of the Linux environment, using a script we'll create later.)
   II. Within this Archive folder, create another folder called "IntyBASIC". (I have this subfolder on one device and not on another, so it's up to you if you want to skip this step.)
   III. Within either the Archive folder or the IntyBASIC subfolder if you created it, create subfolders for each project you are importing.
   IV. Use any means (mini USB stick, cloud storage, etc.) to copy all your files from each project into the appropriate project subfolder.
   V. From each project folder, zip all the files within that folder into a .zip file.

E. Back in Termux, run these commands:
```
mkdir projects
```
```
cd projects
```

F. The "mkdir" command was to "MaKe a DIRectory" and the "cd" command was to Change into that Directory. Now, for each project you are importing:
   I. Use the "mkdir" command to create a subfolder for that project, same as what you did at step 8D-III above with the Archive folder in your Android filesystem. If your project is named "Project", it will look like this:
   ```
   mkdir project
   ```
   II. Go into that subfolder with the command
   ```
   cd project
   ```
   III. Run the command
   ```
   termux-storage-get project.zip
   ```
   and select the zip file for that project from your Android filesystem. It should be in the Archive folder you created at step 8D-I.

(Note: there is no need to rename “project.zip” into a relevant name, but it’s up to you.)

IV. Run the command
```
unzip project.zip
```
V. Run the command
```
cd ..
```
to return to the “projects” folder.

VI. Repeat for any additional projects you are importing.

G. Since we are leaving the IntyBASIC binary in the Home folder, take some time now to modify the INCLUDE statements in your source file(s).

This is a good time to become familiar with your text editor. Install a different one if you prefer.

Because of the limited interchange allowed between the two filesystems, you will want to use one of these text editors instead of one available as a separate Android app.

Step 9: Create your first script: Edit.

Okay, so you've probably recovered by now from the possible shock of having to work in a text-based environment. First of all, that doesn't have to be the case. By the time we're done, you'll have a graphical environment that you can work in if you prefer.

Now for the upside: it's a lot faster to execute commands with a couple keystrokes than to go clicking or tapping on a bunch of different icons. The same goes for doing the editing tasks through keyboard combinations rather than clicking around in various menus.

A. Make sure you are back in your Home folder by entering the command
```
cd
```
B. As a failsafe, enter the command
```
which bash
```
You should see:
```
/data/data/com.termux/files/usr/bin/bash
```
What is BASH? It's the shell running within Linux that interprets your commands. You can substitute it for other shells, but we're not going to do that here.

C. Using your text editor, create a file called "~/e.sh" and enter the following text:
```
#!/data/data/com.termux/files/usr/bin/bash
clear
echo What are you working on?
echo ------------------------
echo "1. Blix & Chocolate Mine"
echo "2. FUBAR"
echo "3. Hunt the Wumpus"
echo "4. Number Zap"
maxchoice=4
read game
case $game in
    1) argsource=projects/bnc
       arglist="$argsource/const.bas
       $argsource/vars.bas $argsource/defs.bas
       $argsource/title.bas $argsource/blix.bas
       $argsource/mine.bas $argsource/graphics.bas
       $argsource/footer.bas $argsource/main.bas"
```

```
        ;;
    2) argsource=projects/fubar
        arglist="$argsource/const.bas
        $argsource/vars.bas $argsource/defs.bas
        $argsource/title.bas $argsource/menu.bas
        $argsource/music.bas $argsource/graphics.bas
        $argsource/footer.bas $argsource/main.bas"

        ;;
    3) argsource=projects/wumpus
        arglist="$argsource/variables.bas
        $argsource/const.bas
        $argsource/instructions.bas $argsource/data.bas
        $argsource/graphics.bas $argsource/footer.bas
        $argsource/main.bas"

        ;;
    4) argsource=projects/numberzap
        arglist="$argsource/const.bas
        $argsource/vars.bas $argsource/defs.bas
        $argsource/title.bas $argsource/menu.bas
        $argsource/music.bas $argsource/graphics.bas
        $argsource/footer.bas $argsource/main.bas"

        ;;
    *) echo Not one of the options.
esac
if [[ -z $game || $game -lt 1 || $game -gt
$maxchoice ]]

then
    echo Goodbye.
else
    emacs $arglist
    echo emacs returned exit status $?
fi
```

D. Save the file and close your text editor.

Time to explain what all this means.

First, the top line needs to be entered exactly as you see it, with no blank line above it. The first two characters, "#!", are often called a shebang or hashbang. The remainder of this line is what you saw earlier from the output of "which bash".

You'll see among the echo statements that some of them contain quotation marks and some don't. In my case, it's because there's an ampersand in one of the game titles ("Blix & Chocolate Mine"), and BASH interprets the ampersand as a control operator.

Note our first use of a variable just below the echo statements, "maxchoice=4". Variable declaration in a BASH script is implicit as opposed to explicit, simply meaning you don't have to declare a variable before you can use it. (You may recall from IntyBASIC development the code "OPTION EXPLICIT ON". That tells the IntyBASIC compiler to generate an error message if it encounters a variable you didn't already declare. That's helpful if you accidentally misspell a variable, and then wonder why you're not getting the expected output.) Just be sure there's no space before or after the equal sign, or you'll get an error. As you might have guessed, change the number appropriately as well.

Next is the "read" statement, which waits for the user to type something, and then saves it to the variable "game". Note there are no data types here. That means we can't guarantee the user enters a number. We check for that later.

To read the value of an existing variable, we preface that variable name with a dollar symbol. We first see that in the Case statement.

Next, we have our first case, with the syntax being the value to look for, followed by a closed parenthesis. Since there are no spaces or other tokens here, we can get away without using quotation marks for our variable assignment to "argsource". This will be the folder path from the Home folder, where all of our scripts are being saved to.

Generating the arglist merely combines all of what you know by now, but note the use of the variable $argsource within a string. Note each case ends with a line containing two semicolons. Below that is "*)", which is the default case if the user input didn't match anything else. Finally, the syntax to close out our Case block is "esac", or "case" spelled backwards.

The syntax for If statements is precise, and be sure to include spaces between all the tokens. The set of conditions we are looking for is encapsulated within double square brackets, and we are using two pipe symbols ("||") as "or". The tokens "-lt" and "-gt" mean "less than" and "greater than", and "-z" tells us if nothing was entered. The rest of the syntax should be obvious, and you might have guessed that "fi" is "if" spelled backwards. The keyword "then" does have to be in a separate line.

Lastly, the variable "$?" gives us the return value from the editor. More on that later.

Step 10: Create your second script: Make.

In case you're not familiar with the term Make, that's a utility within Unix that looks for a file named "makefile" within the current folder and executes the shell commands listed in there.

A. Using your text editor, create a file called "~/m.sh" and enter the following text:

```
#!/data/data/com.termux/files/usr/bin/bash
clear
echo What do you want to Make?
echo ------------------------
echo "1. Blix & Chocolate Mine"
echo "2. FUBAR"
echo "3. Hunt the Wumpus"
echo "4. Number Zap"
maxchoice=4
read game
case $game in
      1) argtitle="Blix & Chocolate Mine"
         argjlp=--jlp
         argsource=projects/bnc/main.bas
         argtarget=projects/bnc/bnc
         argfolder="Blix & Chocolate Mine"
         ;;
      2) argtitle="FUBAR"
         argjlp=--jlp
         argsource=projects/fubar/main.bas
         argtarget=projects/fubar/fubar
         argfolder="FUBAR"
         ;;
      3) argtitle="Hunt the Wumpus"
         argsource=projects/wumpus/main.bas
         argtarget=projects/wumpus/wumpus
         argfolder="Hunt the Wumpus"
```

```
        ;;
    4) argtitle="Number Zap"
        argsource=projects/numberzap/main.bas
        argtarget=projects/numberzap/numberzap
        argfolder="Number Zap"
    *) echo Not one of the options.
esac
if [[ -z $game || $game -lt 1 || $game -gt
$maxchoice ]]

then
    echo Goodbye.
else
    echo Okay, Step 1, Compiling the code.
    echo ------------------------------------------
    ./intybasic_termux -title "$argtitle" $argjlp
    $argsource $argtarget.asm

    var=$?
    echo ------------------------------------------
    echo IntyBASIC returned exit status $var
    if [ $var -ne 0 ]
    then
        echo That means compilation failed.
        Goodbye.
    else
        echo That means compilation was successful.
        echo So, on to Step 2, Creating the ROM
        image.

        echo ------------------------------------
        -----

        echo
        ./as1600 -j $argtarget.smap -s
        $argtarget.sym -l $argtarget.lst -o
        $argtarget.bin $argtarget.asm -m
```

```
                var=$?
                echo
                echo ----------------------------------------
                -----

                echo as1600 returned exit status $var
                if [ $var -ne 0 ]
                then
                    echo That means assembly failed.
                    Goodbye.
                else
                    echo That means assembly was
                    successful.

                    echo So, Final Step, Displaying the
                    Config file,

                    echo and outputting the final product.
                    echo ----------------------------------
                    ----------

                    echo
                    cat $argtarget.cfg
                    cp $argtarget.*
                    "storage/shared/Projects/IntyBASIC/$arg
                    folder"
                fi
            fi
        fi
```

B. Save and close.

Now, for an explanation of what's new here.

You'll notice that some cases, 1 and 2, use the variable "argjlp", while the other cases do not. That variable is later referenced when we invoke intybasic_termux. It's fine if we hadn't implicitly created that variable at that point; it won't affect anything.

You'll also notice that the value of the variable "argtitle" in all cases is in quotes, and then its reference in intybasic_termux is also in quotes. That's because we want the quotes when we run intybasic_termux.

As for the other variables: “argsource” is the path to the file to run compilation on, “argtarget” is the path and filename of the files to be created (the .asm file, the .bin and .cfg file, and all the debugger files), and “argfolder” is the name of the folder within your device’s storage to copy the .bin and .cfg files to. (We will create a separate script for backing up the source files to this same folder later.)

There is no need for me to elaborate on the call to intybasic_termux.

What I’ll point out here is that I save the return value from intybasic_termux (“$?”) to a variable for later use. Why couldn’t we just use “$?” in the If statement below? Because there were a couple of Echo statements after the call to intybasic_termux. Using “$?” here would just give us the return value from “echo”.

As you could probably guess, “-ne” in the If statement means “not equal to”. A return value of 0 means there were no errors from the IntyBASIC compilation. This is how we automatically know whether to move on to the next step or not.

Because I don’t expect you to know what all the as1600 switches mean, we’ll go over that at the next step. It will be a good time to make sure everything up to this point is working anyhow.

Near the bottom, we have “cat”, which is an abbreviation of “concatenate”. It is commonly used in Linux to simply view the contents of a file, which is what we do here.

Finally, we have “cp”, which is an abbreviation of “copy”. We are copying all the files we just generated and putting them into the appropriate project subfolder out in the Android filesystem. Again, you may or may not have created an intermediate folder called “IntyBASIC” so pay attention to that.

Step 11: Create your third script: Backup.

After this, we'll take a break from script-writing and check to make sure everything so far is working. There are still a couple more scripts to write.

A. Using your text editor, create a file called "~/b.sh" and enter the following text:

```
#!/data/data/com.termux/files/usr/bin/bash
clear
echo What do you want to back up?
echo ----------------------------
echo "1. Blix & Chocolate Mine"
echo "2. FUBAR"
echo "3. Hunt the Wumpus"
echo "4. Number Zap"
maxchoice=4
read game
case $game in
      1) argsource=projects/bnc
         argfolder="Blix & Chocolate Mine"
         ;;
      2) argsource=projects/fubar
         argfolder="FUBAR"
         ;;
      3) argsource=projects/wumpus
         argfolder="Hunt the Wumpus"
         ;;
      4) argsource=projects/numberzap
         argfolder="Number Zap"
         ;;
      *) echo Not one of the options.
esac
if [[ -z $game || $game -lt 1 || $game -gt
$maxchoice ]]
```

```
then
    echo Goodbye.
else
    cp $argsource/*.bas
    “storage/shared/Projects/IntyBASIC/$argfolder”

    echo “Backed up source files for $argfolder”
fi
```

B. Save and close.

Step 12: Additional steps

A. Enter the command
```
echo $PATH
```
You should see:
```
/data/data/com.termux/files/usr/bin:/data/data/com.termux/files/usr/bin/applets
```
This is the path, a list of all the folders where BASH looks to find files if you don't specify the folder. That is why putting "./" before filenames had been important this whole time – because the current folder is not in this list.

If you've ever seen "." and ".." in each folder listing before, now you know that "." points to that folder and ".." points to the folder's "parent." Additionally, the tilde character (~) points to your Home folder, and you can always return Home by simply typing "cd" as we did once before. (This is different from MS-DOS, which displays the folder you are currently in if you type "cd".)

Each folder in the list is separated by the colon character, so here we have two folders that BASH is looking at to find a file that you refer to.

(In case you were wondering where the Linux filesystem resided within your device's storage, now you know. It's in /data/data/com.termux/files. By default, this filesystem is hidden to all other Android apps. We'll keep it that way now, but you can change that on your own if you like.)

B. With all that said, the next thing we'll do is modify the path to imply the current folder going forward. Enter the command
```
PATH=.:$PATH
```
To break it down, we're adding the current folder "." at the beginning of whatever the path was before. BASH goes in list-order to find files.

If we had entered "PATH=$PATH:.", putting the current folder at the end of the list, there's a chance we would later refer to a name that exists for a file/subfolder in one of the other folders in the path, and unexpected things could have happened later that are potentially catastrophic.

Be warned that Linux gives you tremendous power that you probably never had before. Creating the Linux environment we just did, with a device still unrooted, keeps that power in check.

C. Now, we're going to make the path change permanent.
   I. Open your text editor and create a file called "~/.bash_profile"
   II. Enter only this line of text:
   ```
   export PATH=.:$PATH
   ```
   There is no need to add a blank line below it.

III. Save the file and close.

D. Just to check, enter the command

```
whereis e
```

You should see:

```
e: /data/data/com.termux/files/home/e.sh
```

- At this point, you might get a message that you have to install another package. It's only 3 megabytes. If you want to skip the "whereis" steps though, that's up to you.
- You can do the same for the other scripts: "whereis m" and "whereis b".

What this means is that the only reference to a file/folder within the path called "e" is the file "e.sh" we had created, and the same for "m.sh" and "b.sh".

I am using the extension ".sh" to indicate a Shell Script. BASH is the name of the shell we are using by default. There are other shells you can shop around for. A script is similar to what you knew to be a Batch File if you have worked in MS-DOS before, but has much more capability.

All I want to point out though for the next step is that in this environment, we would rather keep the keystrokes to a minimum for mundane tasks you will be doing. Rather than type "e.sh" or "./e.sh" at the command line every time you want to edit your project files, typing only "e" would be better.

E. Enter the commands

```
ln -s e.sh e
```

```
ln -s m.sh m
```

```
ln -s b.sh b
```

We just created three Symbolic Links. What this means is that typing "e" will run the script "e.sh", which saves a few keystrokes. The same goes for typing "m" or "b". It might not seem like a big deal now, but it will have been a huge time-saver by the time you complete a project in this environment.

F. Finally, enter the command

```
chmod +x *.sh
```

What is chmod? It is used to set file permissions. Files in Linux can be readable or not, writable or not, and executable or not, for yourself, others in your user group, and everybody else. In this case, we made the Shell Script files executable.

Commonly, people type "chmod 755 " followed by the file/folder(s) specified. This means "readable, writable, and executable for yourself; readable and executable only for everybody else."

This is important because you can't assume that a newly-created file will have the permissions you're probably used to in other environments. One too many times,

I posted a file onto my old website, The Intellivision Library (still graciously hosted here, http://spatula-city.org/~intvlib/inty/, after all these years), only to find out later I forgot to change the file's permissions, and everybody who tried to access the file through my website received a "403 Forbidden" message until I corrected the problem.

Just as Termux colors folders violet and symbolic links cyan, files with executable permissions are colored green. If you want to double-check, then enter "ls" ("LiSt") to get a listing of all the files and folders within the current folder.

If you tried the "ls" command just now, you might have noticed that .bash_profile seems to be missing. That's because files and subfolders beginning with a period are hidden. To see them, type "ls -a" instead.

One more thing I'll mention here. You can type "ls -l" to get more information about each file/folder, such as the current permissions, its owner (who can change the permissions), and so on. If you want to know at a glance whether the permissions are set properly, that is the way to do it.

Step 13: Try it all out so far

Now, for the next moment of Truth. Does this all work? You've got IntyBASIC, as1600, and jzintv. Up to this point, you have an environment where you can edit and compile/assemble your projects. There's more to do to get jzintv running, but for now, let's try running IntyBASIC and as1600.

A. Enter the command
   b
   You should get the following output:
   `What do you want to back up?`
   followed by your list of projects.
B. To begin with, try an invalid number, like 0.
   You should get the following output:
   ```
   Not one of the options.
   Goodbye.
   ```
C. Run the Backup script again. This time, try pressing Enter without inputting a number.
   You should get the same output as before.
D. Run the Backup script again. This time, try a non-numeric input, like "a".
   You should get the same output as before.
E. Run the Backup script once more, with a valid input this time.
   In my example, "1" would back up Blix & Chocolate Mine.
   You should get the acknowledgement, "Backed up source files for Blix & Chocolate Mine" (or whatever your project is called).
F. Before we try the next script, take a moment to verify the IntyBASIC source files did in fact back up into your Projects folder on your device storage. You should see a current date/time for each of the *.bas files.
G. Now, let's try the Edit script:
   e
   You should get the following output:
   `What are you working on?`
   followed by your list of projects.
H. Try all the invalid inputs: 0, a number too high, a letter, and nothing.
I. Now try the Edit script with a valid input.
   Your text editor should open with all of your source files for your project.
J. Close your text editor and try the Make script now.
   m
   You should get the following output:
   `What do you want to Make?`
   followed by your list of projects.
K. Again, try all the invalid inputs: 0, a number too high, a letter, and nothing.
L. Run the Make script with a valid input this time.

The Make script should run all the way through, if there are no errors. If there is an error in compilation, the script will stop after running IntyBASIC. If there is an error in assembly, the script will stop after running as1600.

If you currently have a project that will build and one that won't, try both. Scroll up (you can swipe the touch screen to scroll) to verify the output is correct.

Step 14: Setting up the graphical environment

A. Launch XServer XSDL now.
   You might want to pay attention to the display resolution when it appears.
B. Download the font pack when prompted to do so. You only have to do it once.
C. Don't worry about making any configuration changes for now. Just wait until you see a blue screen with white text.

   This blue screen is actually the "wallpaper" of the graphical display. All you see right now is called the "root window," analogous to the "desktop" in other Operating Systems.

   Like Termux, XServer will continue running until you tell it to stop. When you're ready to stop it, open your device's Notifications, find "XServer XSDL is running," and tap the STOP button.
D. Now, let's get some wallpaper.
   I. If you have to download a .jpg file first, do that now.
   II. Go back to Termux.
   III. From your Home Folder ("~", where you should already be at), enter the command:
   ```
   termux-storage-get wp.jpg
   ```
   IV. Select your file.

Step 15: Configure the Linux environment for graphics

Unlike with the text editor, where I let you choose any one you want, I will specify a couple things for now that you could otherwise choose on your own. Things would get too complicated if I didn't do it this way. Once you get settled, you can shop around for alternatives or configure what you have.

A. Using your text editor, open "~/.bash_profile" again.
B. Add this line at the end:
```
export DISPLAY=localhost:0
```
   - If you read the text in the blue screen, it also mentioned "PULSE_SERVER". I took that out here, because I was able to get sound working on my tablet only after I commented it out.
   - SDL, which jzintv uses, tries to initialize pulseaudio itself for its sound libraries. Defining PULSE_SERVER creates a conflict, resulting in a "no audio device" error from jzintv and no sound.
C. Save and close.
D. Enter these commands:
```
pkg install x11-repo
pkg install man
pkg install aterm
pkg install xorg-twm
cp ../usr/share/X11/twm/system.twmrc ~/.twmrc
pkg install feh
export DISPLAY=localhost:0
feh --bg-max wp.jpg
```

So what are all these commands? First, "x11-repo" makes available a repository for software packages that use X-Server. Second, "man" is short for "manual" and is a program to display "man pages", or manuals for various software packages. Third, "aterm" is a "terminal emulator" that doesn't take up too much space and can be configured to look visually attractive. There are plenty of alternatives you can shop around for later.

Next, "xorg-twm" is a small "window manager" that is old but good enough for our purposes. If shopping around for such a thing seems like a new concept to you and you're familiar with Windows XP, then you may recall having had a choice between "Windows Classic Style" and "Windows XP Style." Similarly, in Windows 10, you can choose between a full-screen Start Menu (as in Windows 8) or something resembling earlier versions of Windows. Those options only let you choose between a couple views; a Window Manager gives you a lot more flexibility than that. To keep it from getting intimidating, I'll hand-hold as before.

The next command made a copy of the default TWM configuration and placed it into your Home folder. That way, we can make a few changes to it.

Next is a very small software package called "feh" which draws wallpaper onto your Root Window. In the two steps after that, we duplicated what we put into .bash_profile, since that will only take effect in future sessions. Finally, running "feh" draws the wallpaper. Take a moment to admire it if you like.

E. Using your text editor, open "~/.twmrc"
F. Add these lines:

```
InterpolateMenuColors

OpaqueMove

RandomPlacement
```

"InterpolateMenuColors" creates a gradient color effect between color definitions within the menu at the bottom of the .twmrc file. Right now, it will probably look ugly to you, but you can come back later and modify the file to make it look nice. Or if you don't want it at all, comment it out by prefacing that line with a '#'.

"OpaqueMove" lets you see the contents of the window as you're dragging it to move, same as the "show window contents while dragging" option in Windows. Believe it or not, way back when twm was developed, only fast computers at the time could redraw the windows fast enough as they were being moved. That's why this option isn't enabled by default. Without this line, you will just see an outline while you are moving windows.

"RandomPlacement" simply places new windows somewhere on the screen, same as what you're probably used to. Without this line, an outline appears and you have to decide where to put each new window that is opened without a geometry parameter (more on that in a moment).

G. Near the bottom of the .twmrc file, at the line that begins with "Aterm", modify the last part of the line to look like this (it's all one line):

```
f.exec "aterm -geometry 160x55+0+0 -tr -trsb -tint gray -fade 75 -tinttype true -fn 6x10 &"
```

So what's all this? First, "geometry" specifies the size of the terminal window in terms of characters: 160 characters wide and 55 characters tall, and all the way against the top left corner of the display (+0+0). The default tiny font that XServer XSDL downloaded earlier will fit this window onto a display with a resolution as small as 1024x768. Why 160 characters wide? Because some of jzintv's debug tools work best at that width. You'll have a chance to increase the font size later.

Next, “tr” will make the background translucent (!), “trsb” will make the scrollbar translucent as well, and “tint gray” (“grey” is not recognized) combined with “tinttype true” will color the window in such a way that it looks truly translucent. We also have “fade 75” which fades the text to 75% when the window doesn’t have focus. This makes it easy enough to quickly tell whether the window is focused or not, but also easy enough to read the text while not focused.
Now we have “fn 6x10” which is the size of each character in pixels. There are a few choices here, but I’m assuming the worst case that you have a lower-resolution display. My tablet is running Android 5.0.1, which is just barely good enough to use Termux at all, and it has a resolution of 1280x800. Chances are, your device’s display resolution is considerably higher than that. The font size choices are: 7x14, 6x10, 6x13, 8x13, and 9x15.

Lastly, “&” is used to make scripts continue to run without waiting for the command to finish executing. In X Windows, that will be important, because if you launch a program through aterm, you might want to run other commands without waiting for that program to finish running.

H. Save and close.
I. Before we move on, if your “exec.bin”, “grom.bin”, and “ecs.bin” files are not all lower-case, then change them to lower-case with these commands:

```
mv EXEC.BIN exec.bin
mv GROM.BIN grom.bin
mv ECS.BIN ecs.bin
```

This is important because, as you may recall, Linux file and folder names are case-sensitive, and jzintv won’t recognize these files if they are capitalized.

Step 16: Create a keyboard mapping for jzintv.

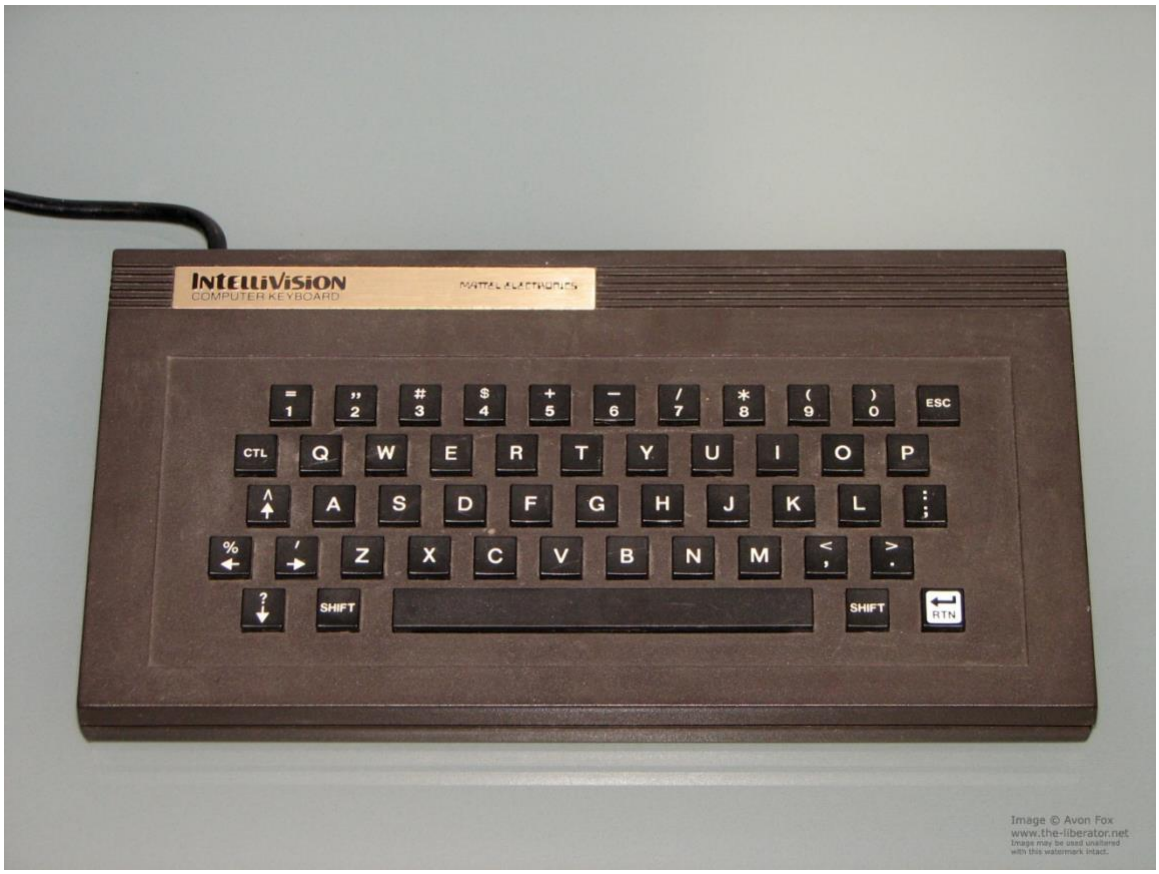
Android keyboards don't have Function keys, which jzintv uses, so it will be necessary to define keybinds for jzintv.

At the very least, we need to have a way to quit when we want to. Resetting and debugging would be good too.

A. Download "hackfile.cfg" onto your device. You can get it from here: http://www.intellivision.us/intvgames/interface/interface.php
Scroll down to the section "How to configure Intellivision's jzIntv" near the bottom of the page, and you will find the link there.
B. Get this file into your Home Folder. Enter the command
`termux-storage-get hackfile.cfg`
and select the file as usual.
C. Open this file in your text editor and make all the necessary changes.

Obviously, it's up to you how to configure it.

Here is an image of an ECS keyboard for your easy reference.



You will notice the following characters missing: "!@&_[]{}\|`~". The three keys just to the right of your keyboard then could be useful for binding some emulator functions.

There are six functions in all that I bound. Four of them I call “PQRS”: PAUSE, QUIT, RESET, and Screenshot (called “SHOT” in the hackfile). The fifth one is BREAK. Starting with P for Pause, I then bound the three keys to the right of P as if they were Q, R, and S. As for Break, I used the backquote key. That’s just my suggestion though.

The sixth function was to toggle between standard controls and ECS controllers, since the default keys to change mappings are Function keys, which Android keyboards don’t have. That required two bindings: one to bind a key to the function “KBD2” (switch to ECS keyboard controls), and another within the third set for the same key to the function “KBD0” (switch back to original controls). You’ll quickly find the third set if you search for “F7” within the file. The second set was pre-configured for the ECS Music Synthesizer.

You can always exit out of jzintv by moving the focus back to the terminal window that you launched it from and pressing Ctrl+C. This is good to know in case you screw up the hackfile or forget to include it when launching jzintv.

This isn’t important, but if you look at the up arrow on the ECS keyboard in the illustration above, you will notice the caret symbol. If you launch an old game in ECS BASIC mode, or if you play Mr. BASIC Meets Bits ‘N’ Bytes, try typing that character into the BASIC interpreter, and you will notice an Up Arrow echo onto the screen. That’s because the Intellivision GROM contains an Up Arrow instead of a caret. Similarly, the underscore is replaced with a Left Arrow. Otherwise, the set of characters 0-94 in the Intellivision’s GROM (Graphics ROM) mirrors the set of characters 32-126 in the standard ASCII table.

Step 17: Create two more scripts: X and Run.

A little more scripting, now that we have our graphical environment ready.

A. Create another file called "~/x.sh" and enter these lines:

```
#!/data/data/com.termux/files/usr/bin/bash
feh --bg-max -z *.jpg
twm &
```

Save and close when you're finished.

What is this script? This is what I use before I enter the graphical environment.

You know what "feh" is now, but it's a little different than before. That -z switch causes feh to choose one file at random among any files ending in ".jpg". Remember, filenames are case-sensitive, so if there are any images you don't want displayed as wallpaper, just capitalize the file extension. Right now, you probably only have "wp.jpg", so that's the only one feh will choose from.

The last line is to launch the window manager, and not to wait before the window manager closes before permitting another command to execute.

B. Create another file called "~r.sh" and enter the following text:

```
#!/data/data/com.termux/files/usr/bin/bash
clear
echo What do you want to play?
echo ------------------------
echo "1. Blix & Chocolate Mine"
echo "2. FUBAR"
echo "3. Hunt the Wumpus"
echo "4. Number Zap"
maxchoice=4
read game
case $game in
    1) argtitle="Blix & Chocolate Mine"
       argsource=projects/bnc/bnc
       argjlp=--jlp-savegame=$argsource.sav
       overlay=$argsource.jpg
       ;;
```

```
        2) argtitle="FUBAR"
           argsource=projects/fubar/fubar
           argjlp=--jlp-savegame=$argsource.sav
           overlay=$argsource.jpg
           ;;
        3) argtitle="Hunt the Wumpus"
           argsource=projects/wumpus/wumpus
           overlay=$argsource.jpg
           ;;
        4) argtitle="Number Zap"
           argsource=projects/numberzap/numberzap
           ;;
        *) echo Not one of the options.
esac
if [[ -z $game || $game -lt 1 || $game -gt
$maxchoice ]]

then
        echo Goodbye.
else
        echo "Okay, $argtitle for Intellivision"
        echo ----------------------------------
        echo 1. No debug, No ECS, NTSC
        echo 2. No debug, No ECS, PAL
        echo 3. No debug, ECS on, NTSC
        echo 4. No debug, ECS on, PAL
        echo 5. Debug on, No ECS, NTSC
        echo 6. Debug on, No ECS, PAL
        echo 7. Debug on, ECS on, NTSC
        echo 8. Debug on, ECS on, PAL
        maxchoice=8
        read var
```

```
if [[ -z $var || $var -lt 1 || $var -gt
$maxchoice ]]

then
     echo Not one of the options.  Goodbye.
else
     if [[ $var -eq 2 || $var -eq 4 || $var -eq 6
     || $var -eq 8 ]]

     then
          argpal=-P
     fi
     if [[ $var -eq 3 || $var -eq 4 || $var -eq 7
     || $var -eq 8 ]]

     then
          argecs=-s1
     else
          argecs=-s0
     fi
     if [ $var -gt 4 ]
     then
          argsdebug="--src-map=$argsource.smap --
          sym-file=$argsource.sym -d"
     else
          argsdebug=-q
     fi
     if [ $overlay ]
     then
          feh --no-fehbg $overlay &
     fi
     jzintv --kbdhackfile=hackfile.cfg -z3 -b4 -
     a48000 $argsdebug $argjlp $argecs $argpal
     $argsource.bin

     var=$?
     if [ $overlay ]
```

```
            then
                kill $(ps aux | grep '[f]eh' | awk
                '{print $1}')
            fi
            echo jzintv returned exit status $var
            if [ $var -eq 0 ]
            then
                echo Thanks for playing.
            else
                echo Better luck next time.
            fi
        fi
fi
```

Save and close when you're finished.

There's only a little bit that needs explaining that you wouldn't already know by now. With each game selection, there will need to be a value for "argtitle" and "argsource". The other two variables, "argjlp" and "overlay", are optional. Copy the "argjlp" line if and only if that game uses JLP (the JLP data will be saved to a .sav file). Copy the "overlay" line if and only if you created an overlay image as a .jpg file.

Here's what's important: before and after the call to jzintv, there is the condition "if [$overlay]". As you know, this condition will evaluate to True if there is a value assigned to overlay. Before starting the game, "feh --no-fehbg $overlay &" opens a new window with your overlay image. The parameter "--no-fehbg" prevents a ~/.fehbg file from being created/overwritten (which we do not use), and since we left out the parameter "--bg-max", it opens in a window instead of becoming wallpaper. You should already know that "&" means to continue executing the script instead of waiting for that window to close.

The call to jzintv does not have an "&" after it, because we want the script to wait until you stop playing. I'll explain some of the jzintv parameters in a moment. Right now, take a look at what happens next. If we opened a separate window with the overlay image, then we want to close it at this point. I could explain exactly what is happening in that "kill" line of code if you could use a nap. The short answer is this: when you started a process using "&", you might have noticed the terminal window display a number. That number is called a PID. We don't know what that number is going to be ahead of time, so the magic within

the line figures it out and then uses that number to close the overlay window. That's all. Just be sure to enter it exactly as you see it.

Now, for the parameters within jzintv. If you want to see the full list yourself, you can enter the command "jzintv --help | more". If jzintv had a man page, we could just enter the command "man jzintv", but it doesn't. You probably already know what "| more" does if you've used MS-DOS, so there's no need to elaborate.

First, we have the hackfile, which is necessary in all cases. Then, I used -z3, which is a good emulator window size between both my devices, 1024x768. -b4 increases the window size by 4 Intellivision pixels per side to draw the border. (I'll tell you later about the graphical environment on my phone, because I did some things differently. This environment laid out here is for my tablet.) Then, we have the audio rate. Curiously, jzintv doesn't run full speed on my phone at 48000 Hz, so I lowered the audio rate to 12000. It does run full speed on my old tablet at the highest rate though. If jzintv doesn't run full speed on your device, try changing this setting.

Now for the options. Depending on user selection, "argsdebug" will either enable the debugger and load the debug files, or else it will do a lot less "whistling". If there was a JLP line, jzintv will run with JLP enabled. Depending once again on user selection, "argecs" will either enable or disable ECS, and "argpal" will either set the A/V mode to PAL or leave it as NTSC. Finally, there's the actual ROM image.

Besides closing the overlay window if there's one open, there's the return value of jzintv. You probably know this, but I'll mention anyhow that your game could misbehave within jzintv, and it will still have a return value of 0 if it exited gracefully. A return value other than 0 will only occur if your game crashes.

Step 18: Try it all out again.

While we're at it, I'll take some time to get you accustomed to twm before you configure it some more.

A. Run the "x" script.
   ```
   x
   ```
   BASH seems to hang. It's actually waiting for you to enter the graphical display before it continues.
B. Switch to the XServer XSDL app.
   You should see your wallpaper image now, if you didn't stop to admire it before.
C. If you have a mouse, hold the primary button. Otherwise, hold your finger on the touch screen.
   A menu should appear. The colors are probably ugly, but we'll change that soon.
D. Drag the mouse pointer to the "Aterm" menu item, and then let go.
   A terminal window should appear in the top left corner of the screen. It should not extend past the right or bottom edges of the screen. Best of all, it should be translucent! The font might be tiny, but you can tweak that. There should also be a "title bar" with icons to the far left and far right (known in X Windows as "window decorations").
E. Try moving the mouse cursor both on and off the aterm window.
   You will notice the title bar "greys out" when the window loses focus, and the cursor changes from a solid box to hollow. Unlike what you might be used to, the mouse cursor must be hovering over a window in order for it to have focus. That's true for twm anyhow. Other window managers might behave differently.
F. Try moving the mouse cursor to the icon on the left and clicking/tapping on it.
   The window should "minimize" (or "iconify" as it's called here), meaning it disappeared and there is a small box with the label "aterm" in it.
G. Try restoring the window by moving the mouse cursor to the "icon" and: if you have a mouse, clicking the secondary button; or else pressing the touch screen (try using two fingers if it doesn't work the first time).
H. Try moving the mouse cursor to the icon on the right and holding it with your finger or the primary mouse button.
   The window should now be in "resize" mode. You should notice a pop-up in the top left corner, showing you the window's geometry as you drag to resize it. Let go when you're finished, of course. You might be wondering at this point where the "close" button is. There isn't one – for now.
I. Try moving the mouse cursor to the title bar and holding it with your finger or the primary mouse button.
   The window should now be in "move" mode. While you are dragging the window, you will notice the portion of the wallpaper seem to move with the window, and snap back once you let go.
   Time to confess: the aterm window is not really translucent. It is fetching the part of the wallpaper that it overlaps and drawing it (with the colors slightly offset) as

its own background image. This will be obvious later, when you raise it on top of another window, and you don't see the other window beneath it where they overlap. Bear in mind, this was quite a few years before the "Aero" effects you have probably seen in Windows 7 or Windows Vista.

J. Now, focus the aterm window and run the "Run" script from here.
r
If the command is not recognized, you might not be in the Home Folder. Just enter "cd" and try again.

The jzintv launch menu we just created should appear within the window. Let's select one of the Debug options (5-8).

The jzintv window should appear, but nothing happens right away. You might need to resize it. We'll handle that in a moment.

K. Move the mouse cursor to put the focus onto the aterm window and enter "q" at the debug prompt.
The jzintv window should close. That was 'q' for "quit."

You should have noticed that, when the aterm window (re)gained focus, it didn't automatically raise on top of the jzintv window. That's also probably different from what you're used to. It will be handy though when using the debugger.

L. Run "r" again (without the debugger this time) and try out your game!
Jzintv should have sound, and it should respond to your key bindings.

Remember, you can focus the aterm window and press Ctrl+C if the Quit button binding doesn't work for some reason.

Step 19: Additional actions

Adding X capabilities to your existing text editor or installing a new one

This depends on your text editor.

For emacs, you can enter the command “pkg install emacs-x”, which will require an additional 60M of storage space. If XServer XSDL isn’t running, emacs will open in text mode as before. You can also force text mode by using the -nw switch.

Trying “man”

You might have been advised at some point to “read the man page for more information” or something like that. There are man pages for aterm and twm in particular, which will be useful when you start changing their configurations.

To read a man page, type “man” followed by the software package. For example, to read the man page for feh, type “man feh”. Try it for man itself too: “man man”.

Use the spacebar to read the next page, or press Enter to scroll one line at a time. Press ‘q’ to quit.

Keeping the text editor open while compiling/assembling

If you choose to do your work in a text environment and you don’t want to keep opening and closing your text editor each time you want to compile and assemble your project, then you can follow these steps to have multiple Termux sessions running concurrently:

A. Swipe from the left edge of the screen.
B. Tap “New Session”.
   You can use one session to keep the text editor open, and the other session to run the “Make” script.
C. To alternate between sessions, swipe from the left and tap the number of the session you want.
D. To close a session, enter the command “exit”.

Tweaking aterm and twm

First, aterm

A. Read the man page for aterm (“man aterm”) and decide on a change you would like to make, say the font size or window geometry.
B. Open the file “~/.twmrc” using your text editor, which may now be graphical.
C. Try making a change to that line near the end beginning with “Aterm”.
D. Save and close.
E. Type “exit” within the aterm window to close it.

F. It should just be you and the wallpaper now. Hold the primary mouse button or your finger on the screen to open the menu, and select "Restart".
   This will apply the change you made to the twm startup script just now.
G. Open the menu as before and select "Aterm".
   The terminal window should look the way you specified. If not, repeat these steps until you like what you see. You might want to keep the terminal window 160 characters wide though, because some of the jzintv debug tools need that.

Now for twm

A. Open the file "~/.twmrc" once again.
   In the bottom section, where the menu is defined, you'll see some lines that contain a pair of color definitions. The first one is the foreground (text color) and the second one is the background. Wherever there are lines without color definitions, twm will color each consecutive line a color between adjacent definitions to create a gradient effect, if the "InterpolateMenuColors" line near the top exists and isn't commented out.
B. Try making some changes here, and then save and close.
C. Restart twm as before. There is no need to close windows. If there are any windows open, their window decorations will simply disappear and reappear.
D. Open the twm menu and decide whether to make some more changes.

Tweaking the Run script

The jzintv window was probably not the right size. Let's change that.

A. I told you earlier about "jzintv -h | more". Type that and press the spacebar a couple times to see what the different "-z" values are. Alternatively, you can set a custom resolution. Note there is no -geometry switch.
B. Open "~/r.sh" and modify the -z and -b values in the eight lines that launch jzintv.
C. Save and close.
D. Type "r" again from the aterm window and choose one of the debug options.
   This will open the jzintv window but not start the game.
E. Focus on the aterm window and type "q" to Quit jzintv.
F. If necessary, repeat these steps until the jzintv window is the desired size.

Experimenting with the jzintv debugger

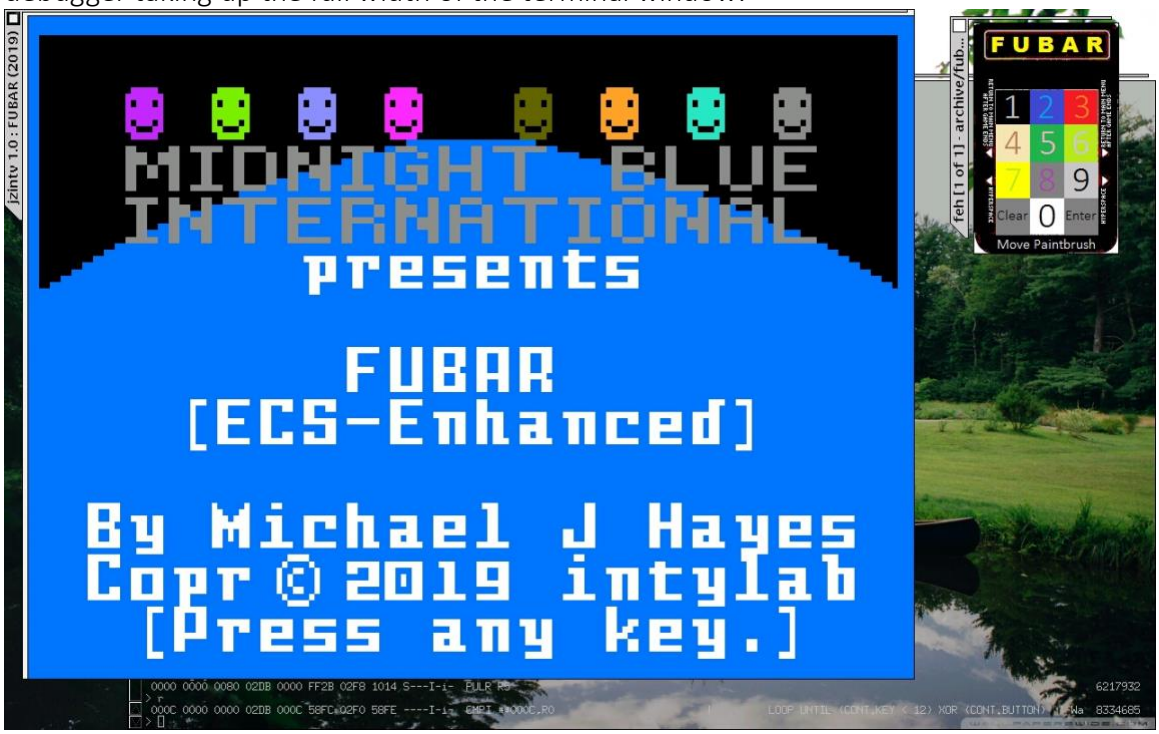
You may or may not be familiar with jzintv's debugger. The additional files generated by the new Make script are not terribly large.

Additionally, I'll show you a couple more features of twm.

A. If necessary, type "m" to remake the final product with the newly-generated debugger files (source map, symbol dump, and listing).
B. Type "r" within the aterm window and choose one of the debug options.

C. Try using the “Raise” selection from the twm menu and select the aterm window to move it on top of the jzintv window.
D. Type “?” to get a listing of debug commands.
E. For now, type “> 160” which tells the debugger that your terminal display is 160 characters wide (jzintv isn’t built to automatically figure that out).
F. Now try using the “Lower” selection from the twm menu and select the aterm window to move it underneath the jzintv window.
G. Keep the focus on the aterm window, and type “r” to start your game.
H. Now put the focus on the jzintv window, and at some point, press the key that you bound to the “Break” function.
I. Put the focus back onto the aterm window, and if necessary, type “?” again.
J. Finally, try some of the commands. As a suggestion, try “m 100” which will give you a partial memory dump of the 8-bit scratchpad RAM (if you haven’t written Intellivision games in Assembly Language before IntyBASIC, $0100 is where your 8-bit variables are stored at). Note the values are all in Hexadecimal (Base-16).
K. You can type “r” to resume the game or “q” to quit.

Here is an illustration from the environment on my tablet. There are some changes I made to my environment subsequent to completing this manual, which I will discuss later. Look carefully in the bottom right corner, and you will see the bottom lines in the debugger taking up the full width of the terminal window.



The only tweak I made to get this screenshot was to modify the Run script, shrinking the border from -b4 to -b2, so you could see that debug line in the terminal window. I’m also using a different Window Manager, wm2, and Aterm is in the bottom right corner.

Automating the jzintv debugger

There's another switch within jzintv that will automatically execute commands within the debugger upon startup. Now that you've experimented with the debugger a little, let's create a jzintv script and enable it.

A. Create a new file with your text editor. I called mine "~/dbscript.txt".
B. Enter the following lines:

```
> 160

r
```

Make sure there's a blank line at the bottom.
C. Save and close.
D. Reopen the Run script ("~/r.sh").
E. In the line where we define "argsdebug" (not as "-q"), change the line to the following:

```
argsdebug="--script=dbscript.txt --src-
map=$argsource.smap --sym-file=$argsource.sym -d"
```

Note there is no space where the line breaks here. You should only have added the --script switch.
F. Save and close.
G. Give it a try.
   I. Launch jzintv with the Run script and try one of the Debug options (5-8). Your game should start on its own, without your having to type "r" first.
   II. Focus the game window and press the key you bound to the BREAK function.
   You should see the bottom line of your aterm window taking up the full width. If you can't see the bottom, click on the title bar of the aterm window to bring it on top.

Creating default settings for Aterm

Don't like launching aterm with all those command-line parameters? Let's create another file so you don't have to do that anymore.

A. Create a new file with your text editor, "~/.Xresources".
B. Enter the following lines:

```
aterm*geometry:160x55+0-0
aterm*transparent:true
aterm*transpscrollbar:true
aterm*fading:75
aterm*tinting:gray
aterm*tintingType:true
```

```
aterm*backgroundType:scale
aterm*font:7x13
aterm*loginShell:true
```

C. Save and close.

Where did all that information come from? The aterm man page. If you want to make some tweaks to this, read the section about "Resources".

Also, that font 7x13 isn't one I listed earlier. Perhaps you'd like to know what all your choices are? There's a package you can install, called "xorg-xlsfonts". Install that and then run "xlsfonts" within Aterm. If you launch it from Termux, you'll have to switch to XServer and back to Termux again for it to run. It lists all your available fonts for X applications.

Appendix A: Future Startup and Shutdown Steps

There are exact steps I follow when I want to stop everything for the day. They may or may not all be necessary, but it's good to get into a habit.

Startup

A. Open the XServer XSDL app.
B. Open the Termux app.
C. Type "x".
   XServer XSDL has to be already running for the "x" script to work. Later, when we return to Termux, there will probably be a bunch of warnings about missing fonts and so on, but who cares? TWM defaulted to something legible.
D. Switch to XServer XSDL.
E. Select "Aterm".

Shutdown

A. Type "exit" from aterm (closing jzintv and anything else first, obviously).
B. Select "Exit" from twm.
C. Return to the Termux app.
D. Type "exit" as many times as there are active Termux sessions (Termux will close when the last session is exited).
E. Swipe down from the top (or however else you open Notifications), find the XServer XSDL notification, and press the Stop button.

Appendix B: "What have I done?"

Since I wrote the original three-volume edition of this document, I have made some changes to the environments on both my devices. You already saw the screenshot from my tablet and might have noticed that twm wasn't the Window Manager anymore.

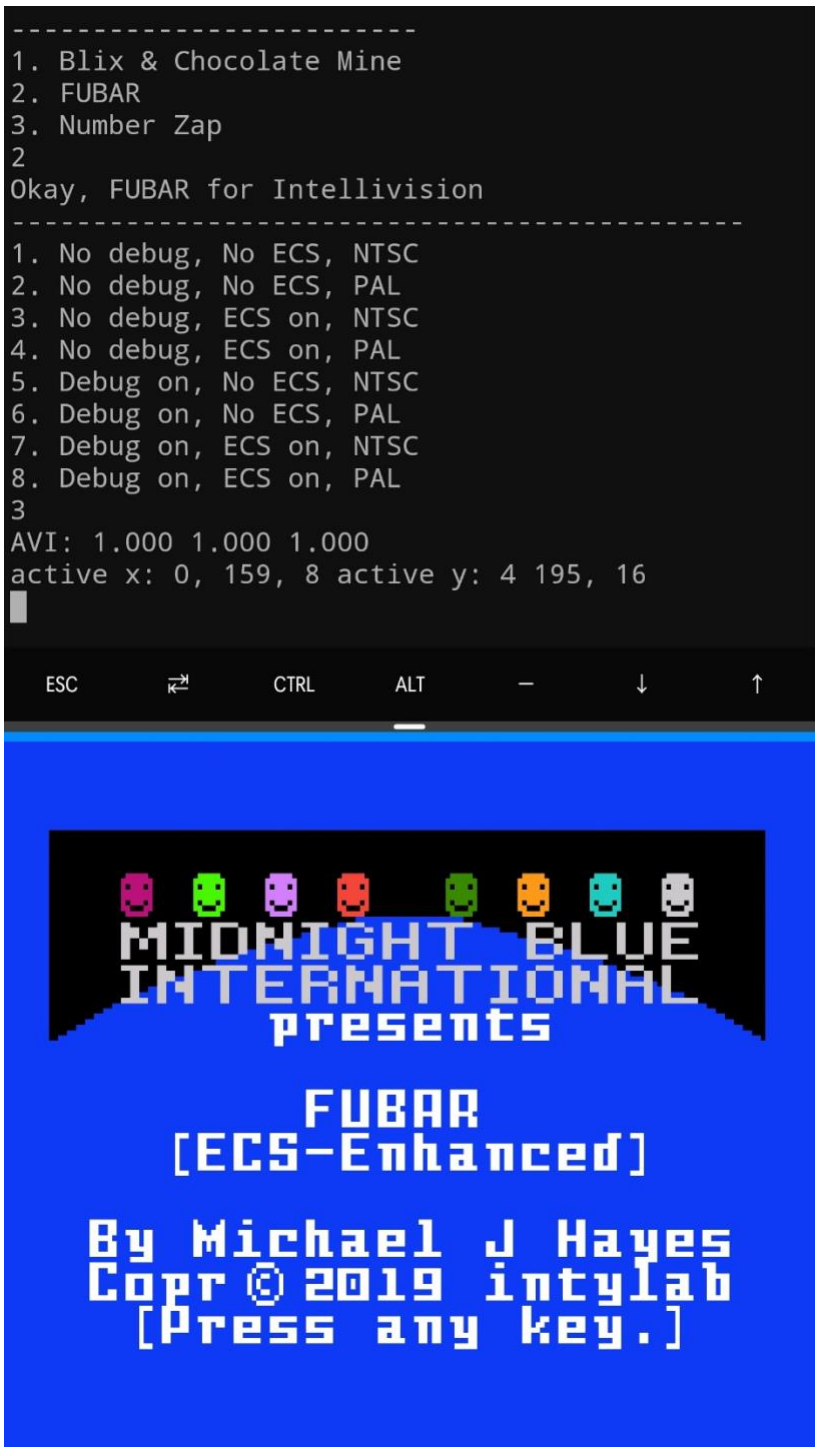
Tablet

- My tablet is running Android 5.0.1 and has only 8GB of internal storage and has a display resolution of 1280x800 in Landscape Orientation. Architecture is "arm". I have a physical keyboard and mouse for this device.
- The font within the aterm window is set to 7x14. That lets me have the aterm display set to 160 characters wide. It's admittedly small, but there's always the Termux session to go back to. The aterm window is just for running scripts anyhow.
- I replaced twm with another Window Manager called Fluxbox. It's one of the recommended Window Managers from the Termux Wiki page. You can go to that page and see what else is recommended, if you want to shop around for something better than TWM.
- To take screenshots, I installed the packages imagemagick and imagemagick-x. That gave me a command called "import". With the command "import -screen ~/shot.png" which I included in "~/.fluxbox/menu", I created a menu-driven command to take that screenshot you saw earlier.

Phone

- My phone is running the latest version of Android and has 32GB of internal storage. The resolution is set to 1080x1920 in Portrait Orientation. It can go higher, but then it consumes more power, and it's really only useful for VR apps. Architecture is "aarch64". I only have a physical keyboard for this device.
- Split-screen works with both Termux and XServer XSDL here, unlike with my tablet, which only allows certain apps to be in split-screen. Between that and not wanting to use the touch screen for mouse support, I made the below changes:
- No Window Manager, and no aterm. The X script just draws wallpaper now.
- I just run emacs in text mode once again. The graphical display is for running jzintv and nothing more.
- I keep the phone in Portrait Orientation, and I launch Termux first, then put it into split-screen mode, and then launch XServer XSDL (which confines itself to the bottom half of the screen and has a resolution of 1080x952).
- In the Run script, I replace the -b switch with -f1, running jzintv in "full screen" within the bottom half of the display. That leaves Termux open at the top for the debugger. All I have to do is touch one half of the screen to focus it.
- This has been the environment I used for my last two projects, from start to end.

Here is what it looks like:



Appendix C: Another Script: Play

It's been all work and no play this whole time. Why not use this environment to kick back and play some Intellivision games?

A. We're going to put all of your Intellivision ROMs and overlay images into a folder on your MicroSD card, assuming you have one.
   I. Open your File Manager app and select your External Storage.
   II. Go into this folder path: "Android/data/com.termux/files"
   III. Create a folder here called "Intellivision".
   IV. Copy all your ROMs and overlay files into this folder.
   V. Make sure all the file extensions are lower-case (the file names before the extension can be mixed-case). Change them if necessary.
   VI. Make sure the overlay image files have the same name as the ROM files.

B. Create another file called "~/p.sh" and enter the following text:

```
#!/data/data/com.termux/files/usr/bin/bash
clear
rompath=storage/external-1/Intellivision
if [ -n "$1" ]
then
    game=$rompath/$1
    if [[ -e $game.bin || -e $game.rom ]]
    then
        echo "$1 for Intellivision"
        echo ------------------------------
        echo 1. No IntelliVoice, No ECS, NTSC
        echo 2. No IntelliVoice, No ECS, PAL
        echo 3. No IntelliVoice, ECS on, NTSC
        echo 4. No IntelliVoice, ECS on, PAL
        echo 5. IntelliVoice on, No ECS, NTSC
        echo 6. IntelliVoice on, No ECS, PAL
        echo 7. IntelliVoice on, ECS on, NTSC
        echo 8. IntelliVoice on, ECS on, PAL
        maxchoice=8
        read var
```

```
if [[ -z $var || $var -lt 1 || $var -gt
$maxchoice ]]
then
     echo Not one of the options.  Goodbye.
else
     if [[ $var -eq 2 || $var -eq 4 || $var
     -eq 6 || $var -eq 8 ]]

     then
          argpal=-P
     fi
     if [[ $var -eq 3 || $var -eq 4 || $var
     -eq 7 || $var -eq 8 ]]

     then
          argecs=-s1
     else
          argecs=-s0
     fi
     if [ $var -gt 4 ]
     then
          argvoice=-v1
     else
          argvoice=-v0
     fi
     if [[ $1 == “Vectron” ]]
     then
          arghackfile=veckeys.cfg
          argscript=veccheat.txt
     else
          arghackfile=hackfile.cfg
          argscript=dbscript.txt
     fi
     if [ -e “$game.bin” ]
```

```
                then
                    romfile="$game.bin"
                else
                    romfile="$game.rom"
                fi
                if [ -e "$game.jpg" ]
                then
                    feh --no-fehbg -g +850+0
                    "$game.jpg" &
                fi
                jzintv --kbdhackfile=$arghackfile --
                script=$argscript -d --
                displaysize=800x600,8 -b4 -a11025
                $argvoice $argecs $argpal $2 $3 $4 $5
                $6 $7 $8 $9 "$romfile"
                echo jzintv returned exit status $?
                if [ -e "$game.jpg" ]
                then
                    kill $(ps aux | grep '[f]eh' | awk
                    '{print $1}')
                fi
            fi
        else
            ls $rompath/*.bin $rompath/*.rom
            echo "$1 not found.  Sorry."
        fi
    else
        ls $rompath/*.bin $rompath/*.rom
        echo Choose from the above list and enter the
        name of the game in quotes.
    fi
```

C. Save and close.

As you probably noticed, there are references to a couple of files we haven't created yet, particularly for Vectron. We'll create those in a moment, but first, an explanation of what's new here.

For "rompath", you'll notice that path "Android/data/com.termux/files" is missing. The folder "external-1" automatically points to that.

Within the first If statement, "-n" lets us know whether you specified a game when running the "p" command. If not, it displays a list of ROMs that you have with a message to choose from the above list. "$1" is the first argument you provide when running a program or script. So to play Vectron, you will type "p Vectron". The quotes around the game title is only necessary when the game has more than one word in the title.

In the second If statement, "-e" checks for the existence of a file. If that file does not exist, the script displays a list of ROMs as before, but with a message that the game you specified is not found.

With that out of the way, we now display the menu you're probably familiar with, but with an exception. This time, the debugger is not an option, but IntelliVoice is. (If you want to modify your Run script for IntelliVoice support and to always keep the debugger enabled, feel free. I just haven't gotten around to writing anything with voice yet.)

The only difference in this call to feh is the geometry. The overlay images I have for commercial games are bigger than the overlay images I created for my homebrew games so far, and with my tablet's resolution being only 1,280 pixels wide, I had to move it over a little. Similarly, I set a custom resolution for jzintv (800x600) with a bit depth of 8 (256 colors).

Skipping all the way down now to the call to jzintv, we have $2, $3, and so on until $9. That allows you to supply additional arguments if you desire. Where that might be handy is if you want to watch a 2-CPU game of Chess or Mind Strike that runs at a normal speed. You can achieve that by disabling Speed Throttling (which means the emulator runs as fast as possible on your device). The switch to disable speed throttling in jzintv is -r0. So for Chess, it would be "p Chess -r0". Remember that for Chess, you press the Enter key on the left controller twice for "CvC", and you still have to make the first move.

With all that said, we're now going to create additional files for Vectron. First is a new keyboard hack file. If you're familiar with Vectron, you probably already know why.

D. Create a file called veckeys.cfg and enter the following text:

```
map 0
1    PD0L_KP1
2    PD0L_KP2
3    PD0L_KP3
```

```
4     PD0L_KP4
5     PD0L_KP5
6     PD0L_KP6
7     PD0L_KP7
8     PD0L_KP8
9     PD0L_KP9
0     PD0L_KP0
UP    PD0L_A_T
LEFT  PD0L_A_L
RIGHT     PD0L_A_R
Z     PD0L_J_WSW
X     PD0L_J_SW
C     PD0L_J_SSW
V     PD0L_J_S
B     PD0L_J_SSE
N     PD0L_J_SE
M     PD0L_J_ESE
P     PAUSE
LEFTBRACKET    QUIT
RIGHTBRACKET   RESET
BACKSLASH SHOT
BACKQUOTE BREAK
```

Be aware, those are zeroes where you see "PD0L".

In case you don't know, Vectron requires all seven directions within the "southern hemisphere" of the disc. Simply using the arrow keys won't work. That is why we created a special keyboard hack file for this game.

There are key bindings for all the numeric keys on the keypad, in case you know about the Easter Egg and want to try to activate it yourself. You probably noticed there are no key bindings here for the second player "PD0R". That's because we're about to create a special debug script which temporarily "patches" the ROM to disable all the enemies, at the expense of also disabling 2-player support.

E. Save and close.
F. Create the other file, "veccheat.txt", and enter the following text:

```
> 160
p 55ed 2b8
p 55ee 4
p 55ef 240
p 55f0 104
r
```

G. Save and close.
H. Like all other scripts, we're going to create a symbolic link and enable its execution. Enter the following at the Termux screen:

```
ln -s p.sh p
chmod 755 p.sh
```

Now it's time to try it all out.

A. Start with just the Play script with no arguments:

```
p
```

You should see a list of all your ROM files (provided they're all in .bin+.cfg or .rom format) followed by a message to choose a game and enter its name in quotes.

B. Next, we'll go with a game you probably have: Astrosmash.

```
p "Astrosmash"
```

If you have the overlay for Astrosmash, it should pop up as before, and if you have the Astrosmash ROM, it should launch. We didn't need quotation marks around Astrosmash since it's only one word, but it's good to be in the habit.

C. Check to make sure you can move left and right, shoot a single shot, and activate Auto Fire and Hyperspace. Use the Quit button when you're finished, as before.
D. Now try Chess, disabling speed throttling as we discussed:

```
p "Chess" -r0
```

After the title screen, press the key bound to the left controller's Enter button twice to set the game to 2-CPU match (you'll see "CvC" in the top left corner). Be aware it's running very fast now. Make the first move, and then the CPU players will take over. The clocks will run very fast, but you can hide them by pressing the right controller's 8 button twice in a row. Watch the match take place at a normal speed now. Press the Quit key when you're done.

E. Now for Vectron.

```
p "Vectron"
```

Remember, we remapped these controls. Use the ZXCVBNM keys for shooting, and the left/right arrow keys to move the Energy Block. By default, the Energy Block doesn't move until you shoot it, unless you press the Up arrow to toggle "Freestyle". Where are the Nasties? We disabled them with that cheat file!

Musings

Here are a few scattered notes that don't have a place anywhere else.

- Yes, I finished FUBAR finally! Three Intellivision cheers!
- I have occasionally launched Termux, only to get a blank screen. If that happens, then long-press anywhere on the touch screen, select "More" and "Kill process" to end the session. Termux will close. Just re-open it and you should be fine.
- If you fall in love with Termux and want to support the cause, you can buy a few add-on modules for $1.99 apiece. One is Termux:Styles, which allows you to tweak the shells' font and color scheme.
- Bear in mind the purpose of this manual. If you know a lot about Linux, I probably insulted your intelligence this whole time, and there are probably a lot of points that are either technically inaccurate or simply false. This is not supposed to be a technical reference. The intended audience is not terribly interested in sweating the details. Do provide suggestions that will make things easier or more failsafe, but don't split hairs over semantics or technical significances.
- This document in its current phase is intended to be just enough to make it possible for somebody to establish a full Intellivision development environment on a mobile device without rooting. Now that I've demonstrated it's possible, we can make it better and easier. It doesn't even have to be Linux-based at all. Ultimately, a whole development and testing environment could be bundled into a single mobile app. Or maybe a Codepen-like website could be used for real-time testing, like what 8bitworkshop.com provides for Atari 2600 development.
- I created this document to allow you to choose any text editor within Termux that you like. But I had to choose one myself, and I chose emacs. At the expense of promoting one environment over another, I will provide files for you that I modified to allow for syntax highlighting for IntyBASIC files. Just ask for them by sending a message to me, Zendocon on the website atariage.com.
- There is another way to have a graphical display besides XServer XSDL. It's called VNC. The Termux Wiki page has information on that. VNC is more secure and requires login, and the setup is a little more complicated.
- If you're using my pre-compiled binary of jzintv, there is one thing that needs pointing out if you run "jzintv -h". Do not use -J as a shortcut to enable JLP. It doesn't work, and I couldn't be bothered to recompile jzintv to fix the documentation.
- I have wm2 installed and running now! It's old and extremely minimalistic, so I don't recommend it, but if you do want it, let me know.
- This document is my contribution to the Intellivision indie scene. I had plenty of help along the way, particularly when trying to install IntyBASIC and jzintv.

Glossary

API: Abbreviation of "Application Programming Interface." A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service. *"Enable Android* API *calls."*

BASH: the default Unix shell included with Linux distributions. It can be replaced with another one if you prefer, but there is no need for us to do that now. *"…* BASH *interprets the ampersand as a control operator."*

Chmod: an abbreviation of "Change Mode." It is a command used to change access permissions to files and folders.

Control Operator: A token (one or more characters with a special significance) that performs a control function. *"… BASH interprets the ampersand as a* control operator*."*

F-Droid: An app repository that offers FOSS (Free and Open Source Software) for Android devices. Applications available here are free to download and modify. Because of its nature, F-Droid itself is not available in the Google Play Store. The official website is https://f-droid.org. *"You may also have installed Termux from* F-Droid*."*

Geometry: in X Windows, the size and position of a window. Most but not all graphical applications allow their dimensions to be defined as a parameter when opening. *"Without this line, an outline appears and you have to decide where to put each new window that is opened without a* geometry *parameter."*

Hashbang: the character sequence "#!" at the beginning of a script. It is called hashbang because the first character is sometimes called a hash, and the exclamation mark is abbreviated "bang". *"The first two characters '#!', are often called a she-bang or* hashbang*."*

Interpolate: in Mathematics, to insert an intermediate value into a series by estimating or calculating it from surrounding known values. *"'*InterpolateMenuColors*' creates a gradient color effect between color definitions within the menu at the bottom of the .twmrc file."*

Keybinds: assignments of key or key combinations on a keyboard with commands. *"Android keyboards don't have Function keys, which jzintv uses, so it will be necessary to define* keybinds *for jzintv."*

Linux: an open-source operating system based on Unix. *"You may not know a darn thing about* Linux *and can't be bothered to 'root' your device."*

Man Page: an abbreviation of "manual page," a document forming part of the documentation of a computer system. *"'man' … is a program to display '*man pages*' or manuals for various software packages."*

Mapping: in Mathematics, a matching process where the points of one set are matched against the points of another set. For emulators such as jzintv, it means defining which keys on the keyboard match up to which functions on the emulated system's controllers. *"Create a keyboard* mapping *for jzintv."*

Path: An environment variable within Linux that tells the shell which folders to search for binaries (executable files) when executing commands. *"This is the* path*, a list of all the folders where BASH looks to find files if you don't specify the folder."*

PID: an abbreviation of Process ID, a number assigned to each process when it is created. We use it here to programmatically terminate a process within a script at a certain point. *"… when you started a process using '&', you might have noticed the terminal window display a number. That number is called a* PID*."*

Repository: a central location in which data is stored and managed. *"First, 'x11-repo' makes available a* repository *for software packages that use X-Server."*

Root: To gain access to the root account of a device such as a smartphone or tablet. It is called "root" because the root account has full access and can perform functions not authorized by the manufacturer or service provider. There is no need for us to do that here. *"You may not know a darn thing about Linux and can't be bothered to '*root*' your device."*

Script: a collection of Linux commands to be executed in sequence. *"Create your first* script*: Edit."*

She-bang: See Hashbang.

Shell: a command-line interpreter that provides a traditional Unix-like command-line user interface. *"[BASH is] the default Unix* shell *included with Linux distributions."*

Shell Script: a program designed to be run by the Linux shell. *"I am using the extension '.sh' to indicate a* Shell Script*."*

Symbolic Link: a type of file that contains a pointer, or reference, to another file or directory. *"Within this folder are multiple* symbolic links*, which Termux colors cyan."*

Window Decorations: the part of a window in most graphical user interfaces that typically consists of a title bar, usually along the top of each window, and a minimal border around the other three sides.

Window Manager: a software utility that manages the overall alignment and layout of graphical windows. It draws "title bars" above all the open windows and allows you to move them around, resize them, and so on. *"'xorg-twm' is a small '*window manager*' that is old but good enough for our purposes."*

X-Server: the part of Linux that allows graphical user interfaces. *"First, 'x11-repo' makes available a repository for software packages that use* X-Server*."*

Acknowledgments

These are in no particular order.

- To Oscar Toledo G., for developing IntyBASIC and revolutionizing Intellivision game development, and also for giving me assistance in getting IntyBASIC to compile in Termux.
- To Joe Zbiciak, for developing jzintv, for creating JLP, and for all the past and present help with everything from my first attempt at installing Linux 20 years ago to getting jzintv compiled and running in Termux. Also for finding the Vectron easter egg and having enough confidence in me to verify it.
- To William Moeller, who gave me enough "inside info" 20 years ago to make my old website The Intellivision Library relevant, and who kept a fire lit under my feet until I finished FUBAR earlier this year.
- To Christopher Neiman, who arranged to put my first two projects onto an Intellivision cartridge (SameGame & Robots), and who also gave me an Intellivision Music Keyboard so I could play Melody Blaster.
- To the developers of Termux for the obvious reason that my "Intellivision Laboratory" is now mobile thanks to them, and also to "xeffyr" for fixing the SDL package so jzintv would play sound.
- To the emacs community for obviating a certain piece of bloatware called MS Visual Studio (2019 was just released – no thanks), and also to "carlsson" on Atariage for his help in getting syntax highlighting to work for IntyBASIC files in emacs.
- To Intellivision Entertainment for creating the (upcoming) Amico console, doing a fantastic job of generating all the excitement about it, and keeping the Intellivision name alive!
- To you, for making it worth my while to have written this document.

The End of this document. The Beginning of your future productivity.