

---

# **GTerm:** A Python Based Telnet Terminal for Unusual Applications

---

Nick Glazzard  
NGIPT

Version 0.8.0  
September 20, 2024

# 1 Introduction

**GTerm** is a simple terminal emulator written in Python that communicates with some operating system or application via the Telnet protocol. It implements a 'glass teletype' style of terminal. There is no way to move the cursor to arbitrary character positions or do other 'VT-100' style tricks.

There are, however, the following potentially useful features:

- The characters displayed for any 7-bit code are defined by the contents of an image file which is used as a texture map. **Any character shape can be accommodated in this way — e.g. APL characters.**
- The texture map is generated from an SVG template with character positions marked out by red lines. By editing this template in Inkscape and inserting the character or drawing you want to get at a location corresponding to a code number, you can have anything you can draw in Inkscape shown as the representation of that code. Tools for making the required texture map from a PNG image file exported from Inkscape are part of **GTerm**.
- **In addition to text, there is a separate 'graphics plane' in which colour vector graphics can be drawn using a simple set of graphics commands.**
- **Characters in the texture map can be chosen to form a virtual keyboard which GTerm can display.** Picking characters on this generates a 7 bit code associated with that character.
- Outgoing characters can be translated in to other single characters or arbitrary strings.
- Similarly, incoming characters or strings can be translated to character codes.
- All drawing is done by Cairo with decent anti-aliasing. Prior to V0.4, it was done primarily by OpenGL with Cairo doing optional anti-aliased drawing. The anti-aliasing was always used in practice, though, and dropping OpenGL for graphics display became an obvious simplification.
- All input/output is logged to a Unicode text log file. Internal character codes can be associated with any Unicode code point.
- Graphics data can be saved to file in SVG format.
- Command line recall and editing is built in to **GTerm**, providing this for systems and applications that otherwise would not have it. This includes CDC NOS APL (see below).
- A very high level graphics facility (in addition to the original graphics scheme) that can be used to plot graphs with almost all the work done by **GTerm**. Again, this is to support the use of APL.

This may seem like a very strange project (and it is!). There is some method in the madness, though. **It was primarily written to serve as an 'application terminal' for an emulated main-frame environment (CDC Cyber machines running NOS 2.8).** Two things were needed for this (beyond the basics):

- Provide a way of displaying colour vector graphics.
- Provide a way of using the APL character set with NOS APL 2. This supports several APL terminals – none of which exist any more in emulation<sup>1</sup>, as far as I can see. It also supports a ‘batch mode’ where APL characters are represented by two letter codes starting with a dollar sign. **GTerm** can map internal character codes associated with APL character glyphs to these strings and vice versa.

Apart from actually wanting to use **GTerm**, it was an opportunity to play with several new (to me) technologies: PyQt, PyOpenGL, Unicode things, PyCairo, Telnet in Python – including not at all well documented things such as option negotiation – and so on. Having the Telnet communications and all aspects of the terminal emulation available in the easily modifiable source code form of a Python program allows for very great flexibility. Just about anything could be added if desired, and many things that would be tedious configuration files can just be changed in the source. (I realize that way of doing things may not be approved of by some, but I like it.)

Were there simpler ways of accomplishing the goals than writing **GTerm**? Well – it might have been possible to modify xterm ...but I’m not sure that would have been the better alternative. In fact, I’m pretty sure it wouldn’t have been. Apart from that, there really didn’t seem to be anything beyond xterm’s Tektronix 4014 emulation as far as graphics went. That is fine – but it is monochrome and the lines are not anti-aliased. There don’t seem to be any other ‘graphics terminal’ emulators out there<sup>2</sup>. Perhaps not surprising at this point in time. As for using NOS APL 2 batch character codes – obviously I was going to have to deal with that myself somehow.

On the other hand, I knew very little about how much support now exists for Unicode before starting this project. There is actually a very great deal – native English speakers probably know a lot less about this than other people. A full set of APL characters is in Unicode – although not in any very sensible order as far as I can see. The internal character codes I chose for these and other characters are totally arbitrary. I could perhaps have made a better attempt at relating these to standards for APL characters and to Unicode. There is some more information on these matters in Section 9.

The first version of **GTerm** was completed in December 2013. The second version added MacOS X support with a simple installer, obviating the need to install many pre-requisites if modification of the source is not required. That version was completed in December 2014. Some modifications were made in March 2017 for PyQt 5 compatibility. Better backspace handling for APL was added in April 2019, followed by extensive changes for line recall and editing as well as the ‘new graphics commands’ in May 2019. Graphics zoom was added in 2023 and text cut and paste in 2024.

For several reasons, the binary installation options were dropped in 2024. This had been implemented using PyInstaller and worked quite well, but the security related behaviour of

<sup>1</sup>True (I believe) in 2013 when this project started – but no longer. See a later footnote.

<sup>2</sup>As of mid-2019, this is no longer true. Rene Richarz has an excellent Tektronix 4014 emulator which reproduces the experience of using that device superbly well. It is compact, clean, self contained, C code. It also includes APL (4015) emulation. <https://github.com/rricharz/Tek4010>

recent versions of MacOS X (now just macOS) made it much less appealing that it originally was. Unless binaries (including shared libraries) are signed (and perhaps more than this now), macOS will perform very time consuming checks (involving ‘phoning home’ to Apple) when the application is run. These can take minutes to perform so there can be a long wait before **GTerm** actually appears!

Consequently, GTerm is now supplied for installation with Python **pip**. In itself, it is a pure Python program, so this is a reasonable approach, but there are pre-requisites which seem to not be fully handled by **pip** and some binary components must be installed using the appropriate system package manager (or Homebrew on macOS).

Please see the current `README.md` file for detailed installation instructions.

## 2 Source Components

This information may be useful to people wanting to modify **GTerm** or adapt it for other purposes. It is not necessary for using **GTerm**. The following files make up **GTerm**:

- `gterm.py`: This is the terminal emulator. It can be run using the command: `gterm` after installation.
- `gtermhostinfo.txt`: This is a list of ‘known hosts’ which can be selected from a list in **GTerm**. The file specifies a name to be shown for the host in the list, the host name or IP address, the port number to use on the host and the host type (`nos`, `nosapl`, `vms`, `unix` or `windows`).
- `mainfont.svg`: This is the template SVG file containing a grid of character positions with fiduciary marks in red and green, together with the default font glyphs in blue. By editing the font glyphs in this file in Inkscape, any character can be drawn for any character code.
- `mainfont.png`: An image file version of `mainfont.svg` exported from Inkscape.
- `getmetrics.py`: This processes the image file `mainfont.png` (by default) and outputs two files: `mainfonttexture.png` (default name) which contains the monochrome texture map for the character glyphs which **GTerm** loads and `mainfonttexture.json` (default) which specifies the texture coordinates of the bottom left of each character glyph in `mainfonttexture.png`, as well as the character width and height in both pixels and normalized texture coordinate units. By default, it also dumps out each character of the font in an image file called `celltest_nnn.png`. These image files can be used by `makevkb.py` to assemble a virtual keyboard image and the metrics needed to locate the selected character on that keyboard.
- `makevkb.py`: This reads a definition file (default: `definitions.json`) specifying the characters wanted on a virtual keyboard and creates an image file containing those character glyphs for use as a texture mapped image in **GTerm**.
- `aplvkb.json`: The virtual keyboard definitions file for an APL keyboard using the default character set in `mainfont.svg/.png`. Feeding this through `makevkb.py` generates `aplvkb.png`.

- `mainfontunicode.py`: This generates a file that maps internal character codes to Unicode code points so that the log file will contain the right Unicode code point for printing. It may be that not all characters have a Unicode equivalent – although the default font does (more or less). Edit this program to set the mapping, which is written to `mainfontunicode.json` (default).
- `klings.wav`: The sound used for the terminal bell. I took this file from Libre Office.
- `build.sh`: A Bash script that can be used to create `mainfonttexture.png` from `mainfont.png`, assemble the APL virtual keyboard image, `aplvkb.png` from `aplvkb.json` and to create a new internal character code to Unicode code point map, `mainfontunicode.json`, from `mainfontunicode.py`.
- `gtermicon.png`: The icon image used in Unity.
- `trygraf.py`: An example class and test program for the graphics commands understood by **GTerm**. Other Python programs could use this to create graphical output to be displayed by **GTerm**.
- `Documents`: A directory containing the documentation for **GTerm** (this document).

### 3 GTerm Controls

**GTerm** is shown in Figure 1. This shows the text display with the APL virtual keyboard being shown. The controls are as follows:

- `To:` This list shows the ‘known hosts’ defined in the `gtermhostinfo.txt` file. Selecting from this list sets the host name, port number and host type controls.
- Host name and Port number text fields. The user can type the desired host name or IP address and port number of the service to connect to in these controls. Selecting a ‘known host’ in the `To:` list fills in these values.
- Host type. This list lets the user select the type of host service being connected to. This sets the appropriate ‘erase’ character, option negotiations on connect, escape sequence handlers and possibly key definitions for that type of host. The choices available are:
  - `Cyber/APL`: Settings for a CDC NOS system with APL 2 batch character sequence and **GTerm** graphics escape handlers. Erase is backspace. Interrupt is `Control-T`. Type `nosapl` in a `gtermhostinfo.txt` host entry selects this host type. Key `F1` is set to output the string: `APL,TT=713`. which will start the APL interpreter in the correct terminal mode.
  - `Cyber`: Settings for a CDC NOS system with **GTerm** graphics escape handlers only. Erase is backspace. Interrupt is `Control-T`. Type `nos` in a `gtermhostinfo.txt` host entry selects this host type. Key `F2` is set to the string: `ABGPLOT, OBEY=APLOT, GET=Y`. which will run the GPLOT plotting program reading commands from `APLOT`.

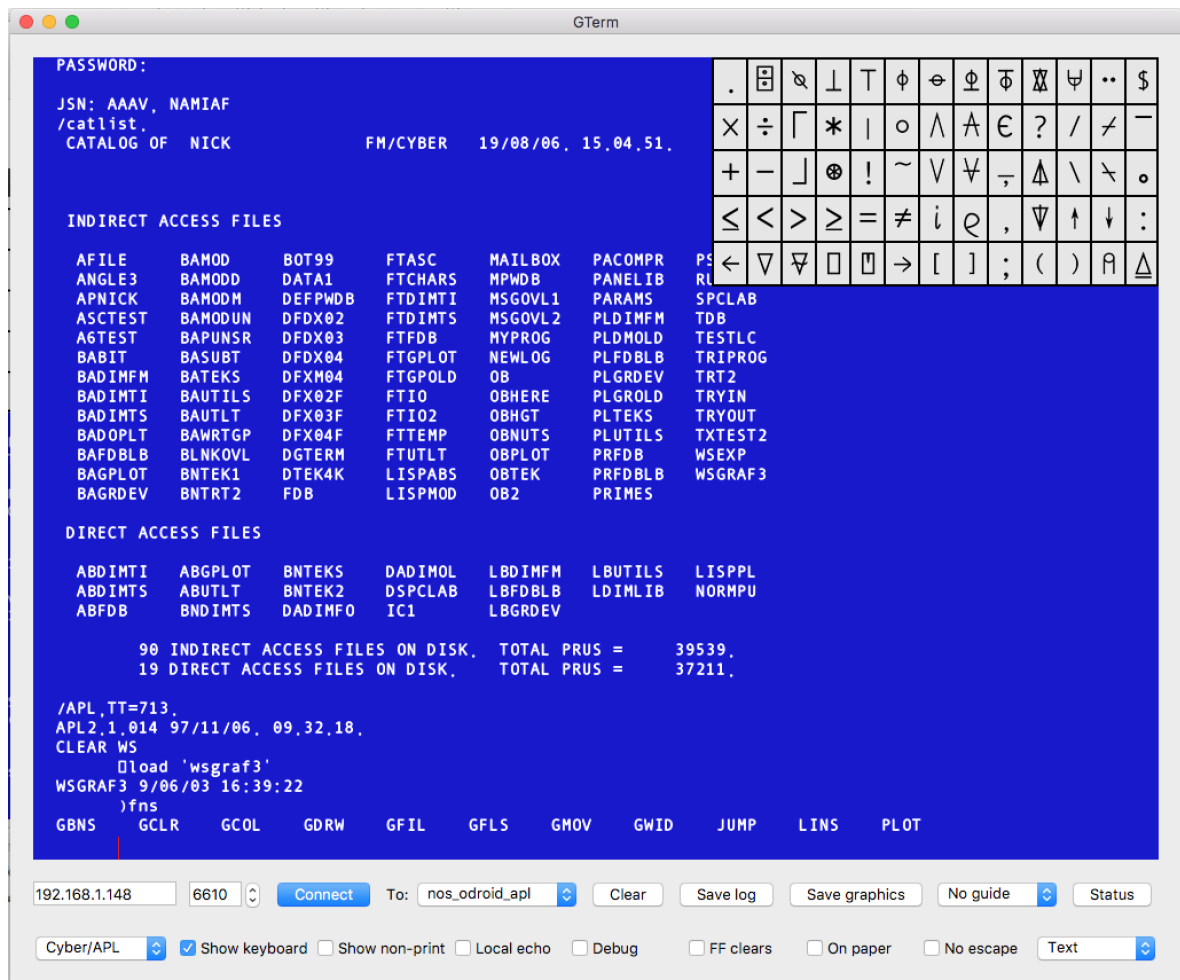


Figure 1: **GTerm** Screenshot

- **VMS:** Settings for a DEC VMS system with **GTerm** graphics escape handlers. Erase is delete. Interrupt is Control-Y. Arrow keys send VT-100 sequences. Type `vms` in a `gtermhostinfo.txt` host entry selects this host type.
- **Unix:** Settings for Unix (including Linux) with **GTerm** graphics escape handlers. Erase is delete. Interrupt is Control-C. Arrow keys send VT-100 sequences. Type `unix` in a `gtermhostinfo.txt` host entry selects this host type.
- **Windows:** Settings for a Windows host with **GTerm** graphics escape handlers. Erase is backspace. Interrupt is Control-C. Arrow keys send VT-100 sequences. Type `windows` in a `gtermhostinfo.txt` host entry selects this host type.

Selecting a 'known host' sets this host type control.

- **Clear** button. Erases the current display (text or graphics).
- **Save Log** button. This pops up a file browser to let you save the session log accumulated so far to a name of your choice. When **GTerm** starts it always opens a Unicode text log file called:

```
/tmp/gterm_log_YYYY_MM_DD_HH_MM_SS.utxt
```

where `YYYY` etc. is the date and time when the log was opened. If a log file is renamed with `SaveLog` a new log file will be re-opened with the current date and time in the name. Note that there are several tools that work well with Unicode text files. The standard Ubuntu text editor (`gedit`) is perhaps the simplest tool to use to view these files ‘nicely’ and to print them. Two tools exist to convert Unicode text files to PostScript: `u2ps` and `paps`. These programs can be used as follows:

```
u2ps gterm.utxt -o mylog.ps
```

```
paps --cpi 20 gterm.utxt > mylog.ps
```

Following which, PDF files can be generated using:

```
ps2pdf mylog.ps mylog.pdf
```

`Emacs` and even `more` in the standard terminal window also work. This degree of support for Unicode text in Ubuntu was a pleasant surprise.

- `Save Graphics` button. This pops up a browser to let you save the contents of the graphics display as an SVG format file. This can be edited in `Inkscape` and processed by many other Ubuntu programs.
- `Guide list` button. This can be used to add column guides to the screen background. Currently, only a Fortran guide is defined.
- `Status` button. This pops up a dialog in which information on the state of **GTerm** is shown.
- `Show keyboard` checkbox. This shows or hides the virtual keyboard (the default being an APL keyboard).
- `Show non-print` checkbox. This turns on the display of non-printing character codes. They are shown on screen as small glyphs containing the hexadecimal value or the standard character name. Note that control characters that are interpreted by **GTerm** (such as LF) will not be displayed.
- `Local echo` checkbox. Echo characters locally. This will rarely be useful except when experimenting with a new type of host, as the host dependent Telnet option negotiation would normally set this automatically if needed.
- `Debug` checkbox. Turns on very extensive debug output to the terminal from which **GTerm** is being run. This usually includes hex dumps of all Telnet traffic. This option is not useful if **GTerm** is started from the Unity launcher, of course.
- `FF clears` checkbox. If set, a form-feed will clear the screen. If not, a line clearly delineating the end of a page will be shown (lots of dashes).
- `On paper` checkbox. Instead of a plain blue background, a simulated green-bar paper background is drawn. This may help with seeing what characters lie on which lines, but is mainly for cosmetic effect.
- `No escape` checkbox. This prevents escape sequences being recognized. This can be useful when trying to write functions that generate such sequences. The characters comprising the sequence will be visible in the editor rather than causing graphics to be drawn.

- **Text/Graphics selector.** Choose to display the text buffer or the graphics buffer. The Clear button applies to the currently displayed buffer.

## 4 Using GTerm

After installation, **GTerm** can be run with the following command from a terminal window (multiple **GTerms** can be run simultaneously):

```
$ gterm
```

**GTerm** may be in one of four states which are shown by the colour of the background of the text window:

- Not connected. Not selected. Background: dark red.  
To connect, select a known host from the `TO:` list **or** enter the *host name or IP address* and *port number* to connect to in the appropriate text fields and select the *host type* from the types list. Then press the `Connect` button.
- Connected. Not selected. Background: dark blue.  
Output from the host will be displayed but keyboard input will not be sent to the host. To select, click in the main terminal text display.
- Connected. Selected. Background: bright blue.  
Keyboard input will be sent to the host when selected.
- Not connected. Selected. Background: bright red.  
This will be the state immediately after logging out of a host.

The colours apply to the paper 'greenbars' in paper mode.

If a command is run on the host that generates graphics output, a count of graphics commands stored so far will be shown in the top right of the text display. E.g. `GC:127` in Figure 1. Switching to the graphics display will show what has been drawn (so far). If no graphics commands have been received, the graphics display will show: `No graphics data.`

Selecting text can be done with the mouse, by left clicking and dragging. The characters in the 2D region so selected will be copied to the window system clipboard.

Additionally, the session log (which is always automatically created when a connection to a host is made) contains all output from the host and all input typed by the user.

The first line of any text in the window system clipboard can be pasted into the input line using `ALT-V` – not `CTRL-V`, as that is a valid character that could be sent to the host.

If a virtual keyboard is displayed, left clicking on a 'key' will enter the associated character. Right clicking on a 'key' will display the 'tool tip' help for that 'key'. This might explain what some peculiar looking characters actually mean!

If the `Ctrl+Alt+ArrowKey` sequence is used on Ubuntu Linux to switch between workspaces, press the `Home` key when focus is regained by **GTerm**. This will reset all internal modifier



Key	Purpose
T	Switch to text view.
G	Switch to graphics view.
K	Toggle virtual keyboard on and off.
U	Scroll text up by 20 lines.
PgUp	Same as U.
D	Scroll text down by 20 lines.
PgDn	Same as D.
H	Turn off any scrolling.
Home	As H, and reset modifiers (see above).
A	Toggle graphics anti-aliasing on and off.
V	Paste printable characters from the first line of any clipboard contents.

Table 1: Hot Keys

key states to the default off state, avoiding possibly insane behaviour if a modifier has been turned on by the workspace switching sequence. Unfortunately, I have not found a way to reliably track the `Ctrl` and `Alt` keys when switching workspaces with `Ctrl+Alt+ArrowKey`.

Also unfortunately, there is no reasonable way to determine the `CapsLock` key *state* in Qt. There are some awful hacks which try to find the X server LED states (which is possible) to get the `CapsLock` key state, but that is both horrible and messy. `CapsLock` *events* are handled correctly, but the initial `CapsLock` *state* is basically unknown (and the best we can do is assume it to be off). This may cause capitalization to be reversed from that expected when switching between applications.

## 5 Hot Keys

A few hot keys are defined which are very useful to avoid losing focus in the main text window when wanting to use some of the more commonly needed controls. The ones currently defined are listed in Table 1. All hot keys use: `ALT+<key>` to activate them (except `PgUp`, `PgDn` and `Home` which work with or without `Alt`).

Scrolling in the text plane is controlled with the mouse wheel (or equivalent). The scroll back buffer is 1040 lines.

### 5.1 Command line editing

The arrow keys are used to perform local command line editing. A buffer of lines input by the user and terminated by `(return)` is maintained inside **GTerm**. These may or may not be ‘commands’, but they will be a superset of them (there is, of course, no way **GTerm** can tell the difference between commands to the operating system, commands (etc.) to applications and data entry). The actual implementation of line editing is pretty tortuous, but it seems to work correctly (finally!).

The `up-arrow` key moves up the command list (further into the past) and the `down-arrow` moves

down the history list towards the present. When the present is reached, a blank line is shown indicating a new command can be entered.

When a history line is selected the `forward-arrow` key moves though the line to the right and the `back-arrow` key moves to right. The current position is shown by a red line cursor positioned between (or before the first and after the last) character on the line. Typing printing characters will insert them at the cursor position. The `backspace` key will delete the character immediately before the cursor (if any).

The `control-A` key combination moves the cursor to the start of the line and `control-E` to the end. The `control-D` key combination deletes the character immediately after the cursor (if any).

The `(return)` key sends the edited (or simply recalled) line to the host.

Note that command line editing works with APL characters as well as ‘normal’ characters.

## 6 Modifying GTerm

As a Python package, all the source of **GTerm** is available for modification. Although there are other options, simply modifying the cloned source and re-installing is one easy way to go.

It is possible to change the glyphs shown for any 7-bit character code by modifying the contents of `mainfont.svg`. The locations for each character code are clearly marked with red lines and anything you can draw (in blue) with an SVG editor can be shown for that code. `getmetrics.py` will generate a texture map for use by **GTerm** from this SVG file.

Editing `definitions.json` (default name) and running `makevkb.py` will assemble a new virtual keyboard image for use in **GTerm**. An example of a definitions file is `aplvkb.json` which shows clearly how this works (including the ‘tool tips’ for each key on the virtual keyboard).

Editing `makefontunicode.py` lets you change the association between internal character codes and the Unicode code point that should be printed for them.

The shell script `build.sh` performs the steps needed to make the files used by **GTerm** from these various primary definition files. The usage text for `getmetrics.py` and `makevkb.py` will show you the available options for these programs.

Since all of **GTerm** is in source code form, almost any modification is possible. The main classes and functions in **GTerm** (in `gterm.py`) are:

- `class XTelnet(Telnet,object):` This extends the standard Python `Telnet` class in various ways.
- `function main_terminal_telnet():` This uses `XTelnet` to implement a complete command line Telnet application which can be run from a Terminal window.

Command	Sequence	Explanation
Clear	<Esc> [ 0 z	Empty the graphics command list.
Colour	<Esc> [ 1 rrr ggg bbb z	Set the current colour to (r,g,b). Range: 0:999
Fill	<Esc> [ 2 z	Fill the graphics window with colour.
Move	<Esc> [ 3 xxxx yyyy z	Move to (x,y) with the 'pen' up. Coordinates: 0:9999
Draw	<Esc> [ 4 xxxx yyyy z	Draw to (x,y) with the 'pen' down.
Flush	<Esc> [ 5 z	Force the graphics command list to be drawn now.
Width	<Esc> [ 6 www z	Set the line width. Range: 1:999

Table 2: Graphics Commands

- `function negot()`: Performs minimal but essential option negotiation on connecting to a host. This aspect of Python's `Telnet` class is not really documented well anywhere – but this code works with the hosts tested so far.
- `class GTermWidget(QGLWidget)`: A PyQt widget that implements all the connection type independent terminal functionality (more or less). Classes derived from this can implement a connection type specific terminal – e.g. `Telnet`. A serial line connection could be another future application.
- `class GTermTelnetWidget(GTermWidget)`: A PyQt widget that adds `Telnet` connectivity to `GTermWidget`.
- `class AudioFile`: Minimal facility to replay `WAV` audio files. Used for the bell!
- functions `cyber_apl_escape()` and `ansi_escape()` process `APL` batch character sequences and `ANSI` escape sequences respectively.
- `class TerminalDialog(QDialog)`: This is **GTerm**'s PyQt GUI class which implements pretty much the whole **GTerm** application using `GTermTelnetWidget` and the usual PyQt widgets. Layout is built in to the program – there are no external 'resource' files.

There are fairly extensive comments in the code which might help with modifications.

## 7 Graphics Commands

These are implemented using a new set of ANSI-like escape sequences. Since these sorts of sequences were being minimally processed to throw them away, it seemed like a convenient place to add graphics commands. The format is not very efficient. It is much more verbose than Tektronix graphics format. On the other hand, it is all readable characters apart from the ANSI Escape character, and it is vastly more concise than SVG seems to be. The *original* (pre-V0.4) commands are shown in Table 2. These are all defined using integers.

Note that there are no spaces in the commands – the spacing between components is just for clarity in this description.

The graphics model is that (2D) graphics commands are sent to **GTerm** and added to a graphics command list. This is drawn when the user switches to the graphics display in

Command	Sequence	Explanation
Clear	@[0 @	Empty the graphics command list.
Colour	@[1 r g b @	Set the current colour to (r,g,b). Range: 0:1
Fill	@[2 @	Fill the graphics window with colour.
Move	@[3 x y @	Move to (x,y) with the 'pen' up. Coordinates: any float.
Draw	@[4 x y @	Draw to (x,y) with the 'pen' down.
Flush	@[5 @	Force the graphics command list to be drawn now.
Width	@[6 w @	Set the line width to w. Range: 0.0 to 9.0
Bounds	@[7 xlo ylo xhi yhi @	Set the bounds to user coordinates (xlo,ylo) to (xhi,yhi).
G-Bounds	@[8 xlo ylo xhi yhi @	Set the graph bounds to user coordinates (xlo,ylo) to (xhi,yhi).
Square	@[G p @	Set 'square bounds mode' (or not). p is 0 or 1.
Text	@[9 s @	Draw text s in the current size and alignment.
TextSize	@[A sz @	Set the text size to sz. 14 is a typical normal size.
TextAlign	@[B al @	Set the text align to al. 0:left, 1:center, 2:right, 3:title.
TextFont	@[C ft @	Set the text font to ft. 0:serif, 1:sans, 2:fixed.
Title	@[E s @	Draw text s as a title for a graph.
Point	@[D x y @	Draw a point at (x,y).
Circle	@[F x y r @	Draw a circle at (x,y), x radius r.

Table 3: New Graphics Commands

**GTerm** and is guaranteed to be fully drawn after a Flush command. The list is emptied by a Clear command.

All commands begin with the usual ANSI escape sequence introducer of <Escape> [ and end in z. Performance is actually quite reasonable.

These original graphics commands were later augmented with a 'higher level' set of commands. These use floating point numbers in the data stream with the idea of moving almost all of the graphics from the host to **GTerm**. They also replace the non-printing escape character with the @ character. These commands are shown in Table 3. In this table, the spaces characters are literal: they need to be there as shown. The original graphics commands are still available and are used by DIMFILM (and hence **GPLOT**).

The difference between Bounds and G-Bounds is that internally the latter will adjust the specified bounds to get a 'nice' range of values with 'nice' 'tick' values for graph plotting.

If Square (with p=1) has been sent *before* Bounds or G-Bounds, the behaviour of those will be changed so that a square in user coordinates will appear as a square when drawn. For G-Bounds, the X range is adjusted so that the tick intervals are the same on both axes and the X range is *centred on* the supplied X range.

The title text alignment centers the text in the display, as opposed to center, which centres the text on the current graphics position. Note that text to be drawn cannot contain the @ character!

## 8 Example Host Code for Graphics Output

Python code to output **GTerm** graphics commands is shown below.

```
class GtermGraphics(object):
    """
    Output GTerm compatible graphics commands.
    Fixed mode is for DIMFILM, basically. The default mode implements something like
    a very simple standalone graphics library rather than a totally dumb device.
    """

    def __init__(self, lun, fixedmode=False):
        self.lun = lun
        self.fixedmode = fixedmode

    def unavailable(self, msg):
        print 'Function: {0}() is unavailable in fixed mode.'.format(msg)

    def clamp(self, v, lo, hi):
        return max(lo, min(v, hi))

    def clear(self):
        """
        Empty the graphics display list. Clear the screen, in effect.
        """
        if self.fixedmode:
            self.lun.write('\033[0z')
        else:
            self.lun.write('@[0@')

    def colour(self, r, g, b):
        """
        Set the drawing colour.
        """
        if self.fixedmode:
            ir = self.clamp(int(999.9*r), 0, 999)
            ig = self.clamp(int(999.9*g), 0, 999)
            ib = self.clamp(int(999.9*b), 0, 999)
            s = '\033[1{0:03d}{1:03d}{2:03d}z'.format(ir, ig, ib)
        else:
            ir = self.clamp(r, 0.0, 1.0)
            ig = self.clamp(g, 0.0, 1.0)
            ib = self.clamp(b, 0.0, 1.0)
            s = '@[1 {0:.3f} {1:.3f} {2:.3f} @'.format(ir, ig, ib)
        self.lun.write(s)

    def erase(self):
        """
        Fill the display with the drawing colour.
        """
        if self.fixedmode:
            self.lun.write('\033[2z')
        else:
            self.lun.write('@[2@')

    def pen(self, x, y, z):
        if z > 0:
            c = 4
        else:
            c = 3
        if self.fixedmode:
            ix = self.clamp(int(9999.9*x), 0, 9999)
            iy = self.clamp(int(9999.9*y), 0, 9999)
            s = '\033[{0:1d}{1:04d}{2:04d}z'.format(c, ix, iy)
        else:
            s = '@[{0} {1} {2} @'.format(c, x, y)
```

```

        self.lun.write(s)

def move(self,x,y):
    """
    Move to user coordinates (x,y). In fixed mode, the user coordinates
    are fixed at (0,0) to (1,1) corresponding to the bottom left and top right.
    In "altmode", these are set by bounds() or gbounds().
    """
    self.pen(x,y,0)

def draw(self,x,y):
    """
    Draw to user coordinates (x,y).
    """
    self.pen(x,y,1)

def flush(self):
    """
    Ensure the contents of the display list are drawn.
    """
    if self.fixedmode:
        self.lun.write('\033[5z')
    else:
        self.lun.write('@[5@')

def width(self,w):
    """
    Set the line drawing width in pixels (as far as possible).
    """
    if self.fixedmode:
        iw = self.clamp(int(99.9*w),0,999)
        s = '\033[6{0:03d}z'.format(iw)
    else:
        iw = self.clamp(w,0.0,9.0)
        s = '@[6 {0} @'.format(iw)
    self.lun.write(s)

def bounds(self,xlo,ylo,xhi,yhi):
    """
    Set up the user coordinate system. Bottom left of the display is at (xlo,ylo)
    and top right is at (xhi,yhi). If square_mode() has been previously issued, the
    *X* bounds will be adjusted so that something that is square in user coords appears
    square in the display.
    """
    if self.fixedmode:
        self.unavailable('bounds')
    else:
        s = '@[7 {0} {1} {2} {3} @'.format(xlo,ylo,xhi,yhi)
        self.lun.write(s)

def gbounds(self,xlo,ylo,xhi,yhi):
    """
    Set the data range for simple graph drawing. The values specified are internally modified
    based on "tick values" to get a generally pleasing range and starting value. If square_mode()
    has previously been used, the *X* range is adjusted so that the tick intervals are the same on both
    axes and the X range is *centered on* the supplied X range.
    """
    if self.fixedmode:
        self.unavailable('gbounds')
    else:
        s = '@[8 {0} {1} {2} {3} @'.format(xlo,ylo,xhi,yhi)
        self.lun.write(s)

def text(self,string):
    """
    Output text at the last move() location.

```

```

"""
if self.fixedmode:
    self.unavailable('text')
else:
    s = '@[9 {0} @'.format(string)
    self.lun.write(s)

def textsize(self,size):
    """
    Set the size of the text in somewhat arbitrary units. 14 is arguably normal size text.
    """
    if self.fixedmode:
        self.unavailable('textsize')
    else:
        size = max(3,size)
        s = '@[A {0} @'.format(size)
        self.lun.write(s)

def textalign(self,alignment):
    """
    Set how subsequent text() is aligned with the move() immediately preceding it.
    alignment 'left' has the text() start there, 'right' has it end there and
    center has it be centered there. 'dispcenter' has it centered in X on the display.
    """
    if self.fixedmode:
        self.unavailable('textalign')
    else:
        aldict = {'left':0,'center':1,'right':2,'dispcenter':3}
        try:
            alcode = aldict[alignment]
        except:
            print 'Unknown alignment name:',alignment
            return
        s = '@[B {0} @'.format(alcode)
        self.lun.write(s)

def textfont(self,fontname):
    """
    Choose a font type (very roughly). Only three choices, as this is not intended
    to be a typesetting solution! 'serif', 'sans' and 'fixed'.
    """
    if self.fixedmode:
        self.unavailable('textfont')
    else:
        fndict = {'serif':0,'sans':1,'fixed':2}
        try:
            fncode = fndict[fontname]
        except:
            print 'Unknown font name:',fontname
            return
        s = '@[C {0} @'.format(fncode)
        self.lun.write(s)

def point(self,x,y):
    """
    Draw a point at user coordinates (x,y).
    """
    if self.fixedmode:
        self.unavailable('point')
    else:
        s = '@[D {0} {1} @'.format(x,y)
        self.lun.write(s)

def title(self,string):
    """
    Draw a graph title in a fixed size and font centered on the display.

```

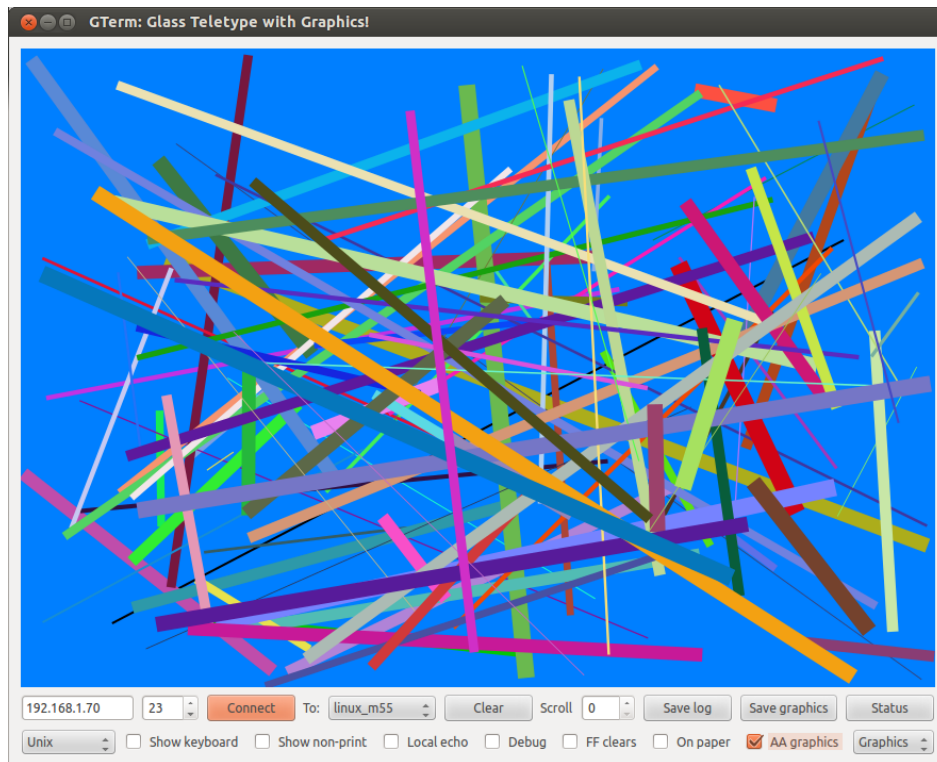


Figure 2: **GTerm** Graphics from Simple Test Program

```

"""
if self.fixedmode:
    self.unavailable('title')
else:
    s = '@[E {0} @'.format(string)
    self.lun.write(s)

def circle(self,x,y,r):
    """
    Draw a circle, center user coords (x,y), radius user X units r. This is always a circle, regardless of
    the bounds set.
    """
    if self.fixedmode:
        self.unavailable('circle')
    else:
        s = '@[F {0} {1} {2} @'.format(x,y,r)
        self.lun.write(s)

def square_bounds(self,yes):
    """
    Modify subsequent bounds() and gbounds() calls so that if a square is drawn in user coordinates
    it appears square on the display.
    """
    if self.fixedmode:
        self.unavailable('square_bounds')
    else:
        iyes = 1 if yes else 0
        s = '@[G {0} @'.format(iyes)
        self.lun.write(s)

```

A screen shot of the **GTerm** graphics display drawn using this library is shown in Figure 2.



## 9 APL and GTerm

In Cyber/APL mode, **GTerm** supports the use of the APL character set with NOS APL 2. The special APL characters are input using the virtual keyboard facility and shown in the display. Log files use Unicode representations of these characters, using the Unicode code points defined in the ISO-IEC/JTC1/SC22 N 3067 "APL Character Repertoire" standard.

Many tools for text editing and display on Linux and macOS 'just work' with these Unicode log files without any special action being required. Unfortunately, this is *not* true of  $\text{\LaTeX}$ . In order to correctly typeset APL source code cut out of log files, the following is needed:

- $\text{\XeTeX}$  or, probably,  $\text{\LuaTeX}$ , must be used. These support Unicode input.
- The font `APL385.ttf` must be installed on the system. This is freely available for download on the Internet.
- Code can then be included in `\verbatim` environments by preceding such sections with `\setmonofont{APL385}`. To switch back to the default 'typewriter' font, use `\setmonofont{AndaleMono}`.

An example of APL source code cut out of a log file and typeset is:

---

```
      ∇Z←A CROSS B
[1]    Z←1ϕ((A×(1ϕB))-(B×(1ϕA)))
      ∇
[2]    ∇
      1 2 3 cross 4 5 6
-3 6 -3
      □SAVE 'APVECT'
APVECT 4/01/04 16:25:50

      A←1 0 0
      B←0 0 1
      A CROSS B
0 -1 0
```

---

The full set of APL characters supported by default with **GTerm** is shown in Tables 4 and 5. These are all the special characters required by NOS APL2 (I believe) along with some 'normal' characters which seemed handy to have on the APL virtual keyboard.

Personally speaking, I think there are better (nicer looking) Unicode characters that match the APL symbols than the ones defined in the standard and implemented in APL385, but there it is. Other APLs require more symbols, I believe (and more could be defined for NOS APL2, such as underlined characters and so on). Although I think it is worth going to all this trouble to use APL as intended, I can quite see how the character set requirements were a major obstacle to APL adoption. The practicalities of using them can be somewhat daunting, even today.

In these tables, I-code is the code number used internally in **GTerm** for the character, U-code is the Unicode code point (note how they are scattered all over the place!), and B-code is the


Symbol	I-Code	Meaning	U-Code	B-Code
.	46	Group/Decimal	002e	—
	184	Matrix Inverse/Divide	2339	\$XD
	154	Transpose	2349	\$TP
$\perp$	150	Base Value	22a5	\$BV
$\tau$	151	Represent	22a4	\$RP
$\phi$	149	Rotate/Reverse	233d	\$RT
$\ominus$	158	1st Coord Rotate/Reverse	2296	\$RU
$\pounds$	152	Execute	234e	\$EV
$\pounds$	153	Format	2355	\$FM
$\blacksquare$	162	Bad Char	25ae	\$BC
$\ddot{\tau}$	164	Escape from Quote-Quad Input	2361	—
¨	166	Break/Dieresis	00a8	\$DI
\$	36	Dollar	0024	—
$\times$	200	Multiply	00d7	\$ML
$\div$	146	Divide	00f7	\$DV
$\lceil$	198	Maximum/Ceiling	2308	\$MX
*	42	Power/Exponential	002a	—
	124	Residue/Magnitude	007c	\$MD
$\circ$	195	Circular Function	25cb	\$CI
$\wedge$	191	And	2228	\$AN
$\tilde{\wedge}$	189	Nand	2372	\$ND
$\epsilon$	183	Membership	220a	\$EP
?	63	Deal/Roll	003f	—
/	47	Compress/Reduce	002f	—
$\nmid$	178	1st Coord Compress/Reduce	233f	\$SM
-	174	Negative Value	00af	\$NG
+	43	Add	002b	—
-	45	Subtract	002d	—
$\rfloor$	197	Minimum/Floor	230b	\$MN
$\otimes$	254	Logarithm	235f	\$LG
!	33	Combinations/Factorial	0021	—
~	126	Not	223c	\$TL
$\vee$	190	Or	2227	\$OR

Table 4: APL Character Set (Part 1)

Symbol	I-Code	Meaning	U-Code	B-Code
⋄	188	Nor	2371	\$NR
⋈	186	1st Coord Join	236a	\$CN
⋈	182	Grade Up	234b	\$UG
⋈	92	Expand/Scan	005c	—
⋈	179	1st Coord Expand/Scan	2340	\$BT
∘	255	Outer Product	2218	\$NL
≤	193	Not Greater Than	2264	\$LE
<	60	Less Than	003c	—
>	62	Greater Than	003e	—
≥	192	Not Less Than	2265	\$GE
=	61	Equal	003d	—
≠	194	Not Equal	2260	\$NE
⌊	185	Index Of/Generator	2373	\$IO
ρ	187	Reshape/Size	2374	\$RO
,	44	Join/Ravel	002c	—
⋈	181	Grade Down	2352	\$DG
↑	177	Take	2191	\$TA
↓	176	Drop	2193	\$DR
:	58	Colon	003a	—
←	160	Specify/Is	2190	\$IS
∇	180	Function Definition	2207	\$DL
⋈	159	Locked Function Definition	236b	\$LD
⌊	156	Input/Output/Distinguished Var/Func	25af	\$QD
⌊	157	Input Literal/Output Same Line	235e	\$QP
→	155	Branch	2192	\$GO
[	91	Indexing/Function Index	005b	—
]	93	Indexing/Function Index	005d	—
;	59	List Separator	003b	—
(	40	Open Paren	0028	—
)	41	Close Paren	0029	—
⌘	161	Comment	235d	\$LP
⋈	163	Line Delete	2359	\$DU

Table 5: APL Character Set (Part 2)

three character string representation used by NOS APL2 in ‘batch’ mode (i.e. when using a terminal that does not natively support an APL character set — I think the only emulator of a terminal that natively supported APL is Rene Richarz’s Tektronix 4015 emulator). The B-codes are what **GTerm** sends to the mainframe when an APL character is ‘typed’, and what the mainframe sends back to **GTerm** to display an APL character.

## 9.1 APL and graphics

Prior to **GTerm** V0.4, the best that could be done was to write out **GPLOT** commands to a file and have **GPLOT** read that file to draw graphs. The new graphics commands make it feasible to output graphics directly from APL, and a workspace has been constructed – **WSPLOT** – which provides the following functions:

- **GCLR** : Empty the graphics commands list.
- **GCOL r g b** : Set the current drawing colour using 0 to 1 range RGB values.
- **GFIL** : Fill the display with the current colour – an erase operation.
- **GFLS** : Flush output to GTerm (not really needed in user code).
- **GBNS xlo ylo xhi yhi** : Set the G-Bounds for subsequent plots. This draws the axes, labels ticks and draws a grid.
- **GNEW**: Prepare to draw a new graph, black on a white background, replacing a tedious **GCLR**, **GCOL**, **GFIL**, **GCOL** sequence.
- **x GPLOT y** : Plot *y* against *x* (equal length vectors). The bounds are automatically set based on the ranges of *x* and *y*. item **x GPLOTS y** : Plot *y* against *x*, but leave the bounds unchanged. This allows multiple plots on the same axes.
- **GMOV x y** : Low level move.
- **GDRW x y** : Low level draw line to.
- **LINS start end step** : Return a vector of length *step* values with numbers evenly spaced from *start* to *end* (inclusive). Modelled on Numpy `linspace()`.

The **WSPLOT** workspace is constructed from APL source using the batch job found in `wsplot.job`. This makes good use of the ‘batch’ mode APL character representations, which may be a good way of storing APL programs external to NOS.

A simple example of graph plotting using these functions is:

---

```

/APL,TT=713.
APL2.1.014 97/11/06. 09.32.18.
CLEAR WS
  □load 'wsplot'
WSPLOT 4/09/20 12:05:03
) fns
GBNS      GCLR      GCOL      GDRW      GFIL      GFLS      GMOV      GNEW      GPLOT      GPLOTS      GWID      LINS

```

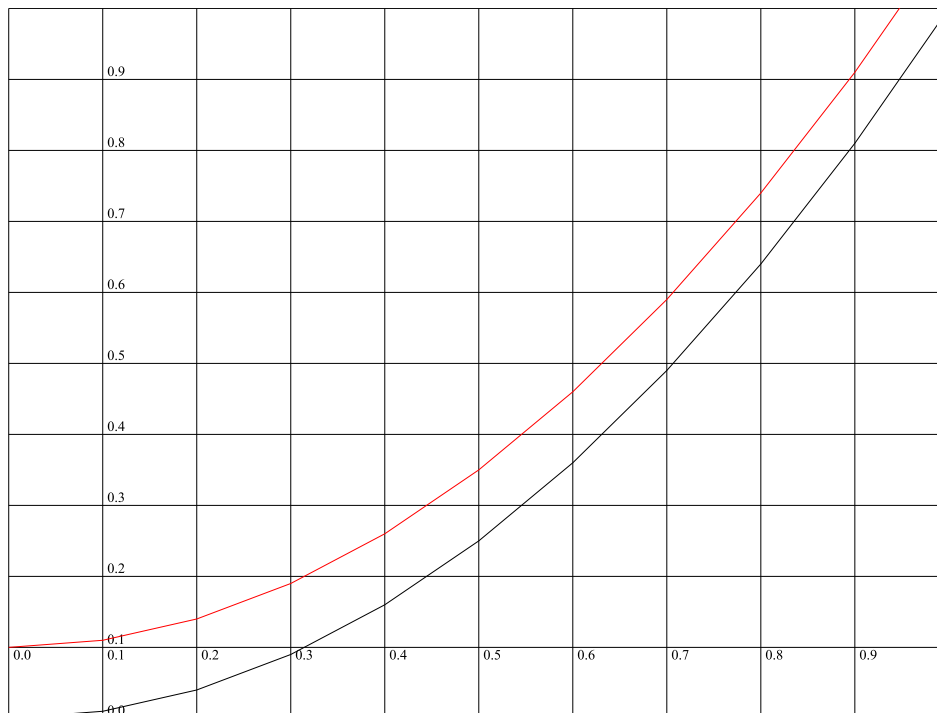


Figure 3: **GTerm** Graphics from NOS APL

```
gnew
x←lins 0 1 11
y←x*2
x gplot y
y←y+0.1
gcol 1 0 0
x gplots y
```

the result of which is shown in Figure 3.

## 10 Data Flow in GTerm

Figure 4 shows the relationships between many of the main components in **GTerm** and how they handle incoming and outgoing characters. This may be of use if modifying **GTerm** — although it isn't fully up-to-date.

## 11 Example `gtermhostinfo.txt` file

This is a simple text file which lists 'known hosts', one per line. An example is:

```
nos_odroid      192.168.1.148 6610 nos
nos_odroid_apl  192.168.1.148 6610 nosapl
```

The space separated fields are:

- Displayed host name shown in the `To:` list.

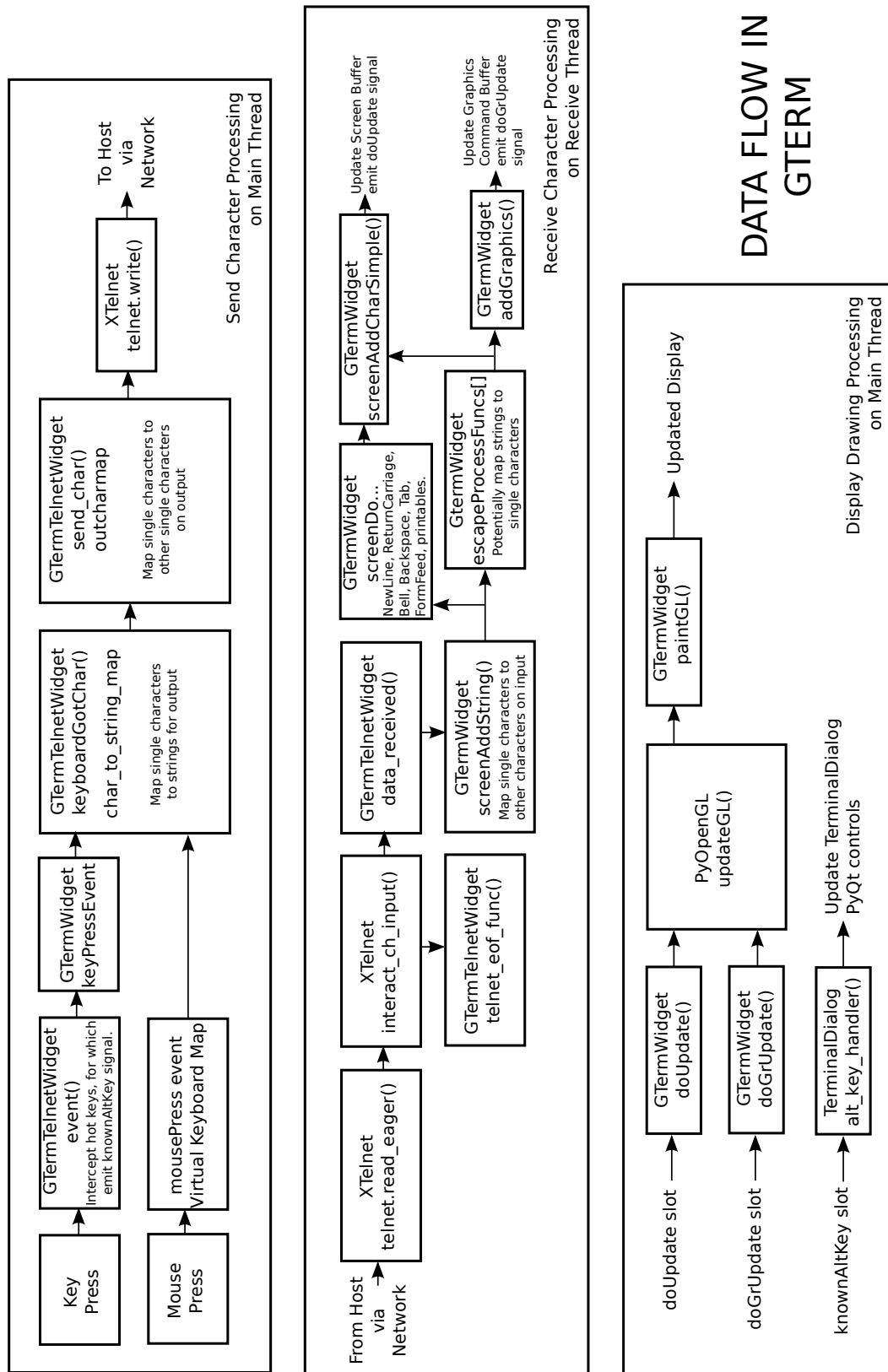


Figure 4: Data Flow in **GTerm**

- IP address of the host.
- Port number on the host.
- Host type name. See Section [3](#) for the options.

The `gtermhostinfo.txt` file lives in the user's home directory.