Image BBS 3.0

Programmer's Reference Guide

Jack Followay, Jr., Ryan Sherwood, Larry Hedman, Al DeRosa

Version 0.4, 2022-09-30

Table of Contents

Chapter 1. Preface

ⓘ Preliminary start on documentation. Feedback welcome!

Welcome to the *Image BBS 3.0 Programmer’s Reference Guide*.

Image BBS has always been a versatile system, allowing a balance between the simplicity of operation and the power of customization with BASIC and ML programming. With the advent of Image BBS v3.0, the power of the system has increased greatly.

Nearly the entire BBS can be customized completely online without the need for programming knowledge. However, Image BBS v3.0 also provides for many powerful new programming tools not before available to the SysOp who desires complete customization of the BBS by changing the BASIC code.

This guide is designed to help such programmers use the Image Application Programming Interface (API) more productively. A lot of hard work has gone into this manual as it was created while the BBS was still being developed. Care was taken to document as much of the BBS as possible for just this purpose. We hope you find this useful in creating the next generation of Image BBS applications.

Happy programming!

—Jack Followay, Jr.; Larry Hedman, Al DeRosa, and Ryan Sherwood

Additional thanks go to jam and X-Tec for editing help, and Jay Campey for pointing out several outright errors in documentation.

Chapter 2. Introduction

Welcome to the first official New Image BBS Programmer’s Reference Guide. While there was an unofficial guide released for Image v1.2, we hoped to give SysOps something they could trust when questions arose about v3.0. Nearly everyone who helped write Image v3.0 helped contribute to this guide. I would like to personally thank everyone who was involved, but if I’ve forgotten anyone please forgive me!

Al DeRosa (Bucko)

Larry Gartin (Wheelman)

Ray Kelm (Professor)

John Moore (Little John)

Ryan Powers (MonOp)

Bob Sisko (Iron Axe)

Ed Wilson (Fred Krueger)

And most of all, I’d like to thank our dear friend, Fred Dart, to whom we owe the very existence of Image v2.0. Without Fred, the project would have been lost forever. It is in his memory that we dedicate Image BBS v3.0.

2.1. How to Use This Guide

2.2. How to Get Help

Chapter 3. Programming Image BBS 3.0

3.1. New Features

`variable=usr(0)` function reads the stack pointer to display how much free space is on the processor stack.

3.1.1. BASIC Keyword Extensions

A few BASIC keywords have had additional parameters added to them.

`load` Relocate Flag

Regular Commodore DOS uses the `,1` in `load"filename",8,1` to mean “load this file to the load address stored in the file.”

In Image BBS, the *relocate_flag* has an expanded purpose—specifying the *segment* of memory to load a particular file to. A segment is the address which a file will load into, ignoring the file’s existing load address contained in the first two bytes of the file.

Currently, some segments are undefined.

Segment	Purpose	Address	Example File
2	Protocol start	`$c000`	`++ punter`
7	Print mode table	x	`e.printmodes`
8	Lightbar table	x	`e.lightdefs`
9	Alarm table	x	`e.alarms`
10	ASCII → Commodore translation table	x	*n/a*
11	Commodore → ASCII translation table	x	*n/a*
12	Network alarm table	x	`nm.times`

Example 1. Loading a Protocol

Here are three different methods of accomplishing the same thing, loading the Punter file transfer protocol.

The first uses the enhanced `load` function:

```
3000 dr=5:gosub 3:a$="++ punter":load dr$+a$,dv%,2 ①
```

And the second relies more heavily on the Image ML, using the parameters set by BASIC:

```
3000 dr=5:gosub 3:a$=dr$+"++ punter":&,7,2 ②
```

In both examples:

- `dr=5:gosub 3` takes an Image drive number (`dr`, 1-6) to return the device (`dv%`, *e.g.* `10`) and drive prefix (`dr$`, *e.g.*, `0:`). Here, it returns the device and drive which your modules are stored on.
- `a$="++ punter"` sets the filename to `load`.

Next:

① Method 1: `load dr$+a$,dv%,2` loads drive prefix `dr$` plus filename `a$` from device `dv%` to segment 2 (`$c000`)

② Method 2: `&,7,2` uses `dr$`, `a$` and `dv%` as defined by BASIC, but is shorter code

The third (preferred) method calls a routine in `im`:

```
3000 a$="punter":gosub 28
```

This method saves time in loading `++` files. If the file specified in `a$` is already in memory, the load process in line 29 is skipped. It also is a shorter syntax for loading a protocol, saving memory in the BASIC program.

`new` Line Range

`new line_number` prepares to `load` a new module by erasing lines from `line_number` to the end of the program.

Example:

`new 3000` erases lines 3000 to the end of all loaded modules.

Hexadecimal Values

Hexadecimal values (base 16) are prefaced with `$`. They can be used in `poke`, `peek`, `sys` and `&` commands. `poke $d004,$0c` is the equivalent of `poke 53252,12`.

Binary Values

Binary values (base 2) are prefaced with `%` and up to 8 bits can be specified. Leading `0`s do not have to be specified.

Binary values can be used with the same BASIC keywords as above. Hexadecimal and binary can be combined, *i.e.*:

```
poke $d004,%00001100
```

is again the equivalent of `poke 53252,12`. Since only the lowest 4 bits are being set, the example can be shortened to:

```
poke $d004,%1100
```

Parsing Limitations

If you want to do logical operations like `and` or `or` with the new binary or hexadecimal prefixes, the only keywords they work with are `poke`, `peek`, `sys`, and `&`.

Putting hexadecimal or binary prefixes after logical operators results in a `?syntax error`:

```
poke $d004,peek($d004) and %1100
```

Instead, assign `%1100` to a variable first, then perform the logical operation using the variable:

```
c=%1100:poke $d004,peek($d004) and c
```

3.2. Programming Etiquette

Be nice. Write clean code. FIXME

3.2.1. Programming Theory

Static vs. Dynamic String Variables

There are two types of strings in BASIC: *static* and *dynamic*.

Static strings look like:

```
10 a$="hello":b$="there"
```

They are given a *string descriptor* (the address, length and type of string) within the BASIC program itself. Normally this isn't a problem. However, when you load another module over the top of that string text, the string pointer still points to the same address. Then when printed, the string variable displays random bytes of the new module, often BASIC tokens.

There's a way to get around this.

Dynamic strings are different: they're created when two strings get concatenated together. You've probably looked through Image's code and seen lines like:

i.SB

```
3192 ... sy$="Su"+"b" ...
```

The reason for concatenating two strings together like this is because strings created this way have string descriptors which point higher in memory (they get created from the top of BASIC downwards). Each new string defined by concatenation creates a new string, moving the pointer to that string downwards, towards the end of your BASIC program—and your static strings.

> This is the whole reason *garbage collection* exists—as more static strings and dynamic strings get defined, the pointers to the next free byte move toward each other. You'll eventually run out of space, triggering a garbage collection as the old definitions of strings get erased to reclaim memory. (Image BBS uses a custom garbage collection routine, which is much faster than the one in standard Microsoft BASIC.)

But the long and the short of it is that the string definition caused by, *e.g.*: `sy$="S"+"ub"` lives higher in memory and the text shouldn't be overwritten by loading another module, causing strings to output bytes of your program.

3.2.2. Module Structure

When developing a module, bugs are certain to happen. These bugs are logged to `e.errlog`. There are variables shown in the error log which can be handy to show which line or module the program was in when it crashed.

Table 1. `e.errlog` Filename Hints

Filename Prefix	Starting Line	Variable Name	Label
`i.*`	3000	`pr$`	Program
`i/*`	4000	`p1$`	Module
`sub.*`	60000	`p2$`	Sub-Module

> `pr$` is considered the main program name. `p1$` is considered a loadable module name (similar to the Little Modem Program *e.g.*, `+.XX.YY`) sub-modules of Image 1.2.

If your module spans from, say, lines `3000-4000` (or beyond), it would be a good idea to do something like:

```
pr$="i."+"test calls":p1$=pr$
```

to assign both `pr$` (error lines 3000-) and `p1$` (error lines 4000-) to the same string. This avoids leaving garbage strings in `e.errlog` because of the previously-mentioned "static string" issue.

If your module is a collection of sub-modules, it is helpful to define in your main program:

```
3000 pr$="i."+"test calls"
```

And in your sub-module, define:

```
4000 p1$=pr$+".structs"
```

so that `e.errlog` reads:

```
Module: i.test calls
Sub Module: i.test calls.structs
```

3.3. Conversion and Backward Compatibility

Image v3.0 is a major change in design and programming style from previous versions. Your first reaction might be to assume that you can no longer use legacy modules created under Image v1.2a and below. However, nearly everything written under previous versions of Image can be made to run under v3.0. The easiest method for this is to run under “1.3 emulation.”

If modules are renumbered to start at line 3000, they can run under Image 3.0.

This mode greatly reduces available memory, but allows a program to make the same calls to `im` that were available under Image 1.2a. The best method for using legacy modules is to convert them to v3.0 compliant modules.

This section is provided to help you as you attempt to convert such modules. The v1.2a ←→ v3.0 cross-reference will be your best tool, but please take time to read the notes regarding making the modules truly Image v3.0 compliant rather than just “compatible.” You will want to take careful note of the theory and structure of a v3.0 compliant program, found in the Programming Etiquette section of this guide.

3.4. `im` Subroutines

This chapter documents `im` routines.

Some routines are simply needed too often by too many modules to justify placing them in every single module. Subroutines are a programmer’s greatest friend. The core `im` file contains many subroutines which you will find basic to your application or module’s needs. Be sure to read the Programming Etiquette section for help when deciding where to place a new routine you create.

> ⇒ Contributions by Jack “Rascal” Followay, Jr; Larry “X-Tec” Hedman, Jay “x” Campey, and Ryan “Pinacolada” Sherwood.

Table 2. `im` routines

v3.0	v1.2, v2.0	Purpose
`goto 0`	`goto 1812`	Jump to main prompt at line 300.

<table>
<tr><th>v3.0</th><th>v1.2, v2.0</th><th>Purpose</th></tr>
<tr><td>`gosub 1`</td><td>`gosub 1001`</td><td>Position record pointer of the currently opened REL file on LFN 2 (it calls line `10`, the position command, twice).
There is a 1/10th-second delay (`&,22,1`) between positioning commands. This is to give a physical disk drive's read/write head (or a slower filesystem, such as on a SD card) time to move to the correct spot within the file.
Setup:
`x`: record number desired</td></tr>
<tr><td>`gosub 2`</td><td>`gosub 1009`</td><td>Sets active system device/drive according to configured setup.
Setup:
`dr`: system drive (`1-6`) desired:
<table>
<tr><th>`dr`</th><th>Purpose</th><th>Prefix</th></tr>
<tr><td>1</td><td>System files</td><td>`s.`, `n.`</td></tr>
<tr><td>2</td><td>E-Mail/NetMail</td><td>`m.`, `nm.`</td></tr>
<tr><td>3</td><td>Etcetera Files</td><td>`e.`</td></tr>
<tr><td>4</td><td>Directory Files</td><td>`d.`</td></tr>
<tr><td>5</td><td>Plus Files</td><td>`i.`, `i/`, `sub.`, `++`</td></tr>
<tr><td>6</td><td>User Files</td><td>`u.`</td></tr>
</table>
Returns:
`dv%`: Active device
`dr%`: Active drive + `:`</td></tr>
<tr><td>`gosub 3`</td><td>`gosub 1010`</td><td>Closes, then re-opens command channel (LFN 15) on the device/drive requested by setting the variable `dr` explained above.</td></tr>
<tr><td>`gosub 4`</td><td>`gosub 1011`</td><td>Opens filename stored in `a$` on device/drive `dr` requested by setting the variable as explained in line 2. Will automatically make call to lines 2 and 3. Upon exit, this routine falls through to line 5.</td></tr>
<tr><td>`gosub 5`</td><td>`gosub 1012`</td><td>Read error channel of currently active device/drive.
Returns:
Upon exit, `a$` contains the error status message, `e%` returns the error number, `e$` the error text, `t%` the track, and `s%` the sector.</td></tr>
</table>

<table>
<tr><th>v3.0</th><th>v1.2, v2.0</th><th>Purpose</th></tr>
<tr><td><code>gosub 6</code></td><td><code>gosub 1006</code></td><td>Uppercase input routine. Waits for user input followed by a carriage return. Stacked commands (<code>uu$</code>, <em>e.g.</em> <code>SB4^R3^Q</code>) are checked for. If any are pending, <code>gosub 310</code> handles checking for displaying command history (<code>^?</code>) or executing a command in the history (<code>^</code> and a digit).
<em>Setup:</em> <code>p$</code>: Text of prompt.
<code>poke 53252,x</code>: Number of characters allowed.
<em>Returns:</em>
<code>an$</code>: the string input by the user.</td></tr>
<tr><td><code>gosub 7</code></td><td><code>gosub 1007</code></td><td>Wait for a keypress.
Returns: an$, a single uppercase character.</td></tr>
<tr><td><code>gosub 9</code></td><td><code>gosub 1360</code></td><td>Prints (strictly text) contents of <code>cm$</code> to <code>Area</code> window of the SysOp screen, unless a network transfer is active (<code>nt=1</code>).</td></tr>
<tr><td><code>gosub 10</code></td><td><code>gosub 1002</code></td><td>Position RELative file record pointer DOS command. Same as line 1. However, it is suggested that you call this routine from line 1 for some systems' compatibility.</td></tr>
<tr><td><code>gosub 11</code></td><td><code>gosub 1004</code></td><td>Each user has a total of 19 flags stored in <code>fl$</code>, which decide whether or not they have access to a particular function or area of the system. These flags are held in a twenty-digit string. Each digit represents a separate flag which can be checked by setting the variable <code>a</code> to the flag number 1-20 (with the exception of 19), and issuing a <code>gosub 11</code> command. If <code>a</code> returns with a value of zero, access is denied, or the flag's value is zero. The values of each flag are as follows:
<table>
<tr><th>Flag</th><th>Purpose</th><th>Flag</th><th>Purpose</th></tr>
<tr><td>1</td><td>Non-Weed Status</td><td>11</td><td>BAR/Log View Access</td></tr>
<tr><td>2</td><td>Credit Ratio</td><td>12</td><td>Sub Maint Access</td></tr>
<tr><td>3</td><td>Local Maint Access</td><td>13</td><td>Files Maint Access</td></tr>
<tr><td>4</td><td>Post/Respond Access</td><td>14</td><td>MCI Access</td></tr>
<tr><td>5</td><td>U/D Access</td><td>15</td><td>Prime Time UD Access</td></tr>
<tr><td>6</td><td>Max. Editor Lines</td><td>16</td><td>Max. Idle Time</td></tr>
<tr><td>7</td><td>Unlimited D/Ls</td><td>17</td><td>Calls Per Day Allowed</td></tr>
<tr><td>8</td><td>Remote Maint Access</td><td><strong>18</strong></td><td>Time/Day Allowed (first digit)</td></tr>
<tr><td>9</td><td>E-Mail Access</td><td><strong>19</strong></td><td>Time/Day Allowed (second digit)</td></tr>
<tr><td>10</td><td>User List Access</td><td>20</td><td>D/Ls per Call Allowed</td></tr>
</table></td></tr>
<tr><td><code>gosub 12</code></td><td><code>gosub 1914</code></td><td>Reset routine. Resets all system output to default parameters.</td></tr>
</table>

<table>
<tr><th>v3.0</th><th>v1.2, v2.0</th><th>Purpose</th></tr>
<tr><td><code>gosub 13</code></td><td><code>gosub 1075</code></td><td>Clear the screen and fall through to line <code>14</code>.</td></tr>
<tr><td><code>gosub 14</code></td><td><code>gosub 1076</code></td><td>Outputs the SEQ file in <code>a$</code> on device, drive <code>dr</code> to the SysOp screen and to the modem.</td></tr>
<tr><td><code>gosub 16</code></td><td><code>gosub 1025</code></td><td>Update Board Activity Register (BAR) statistic <code>x</code>.
This routine:
1. inputs <code>st(x)</code> from record <code>x</code> of <code>e.stats</code> file
2. adds the value of <code>i</code> to it (which can be negative if you want to subtract from the statistic)
3. falls through to line 17.
Setup: open <code>e.stats</code> (<code>gosub 30</code>).
<code>x</code>: the desired statistic to be updated:
<table>
<tr><th>Description</th><th>Last</th><th>Log</th><th>Current</th><th>Total</th></tr>
<tr><td>Feedback</td><td>1</td><td>12</td><td>23</td><td>30</td></tr>
<tr><td>SysOp Mail</td><td>2</td><td>13</td><td>24</td><td>31</td></tr>
<tr><td>User Mail</td><td>3</td><td>14</td><td>25</td><td>32</td></tr>
<tr><td>Posts</td><td>4</td><td>15</td><td>26</td><td>33</td></tr>
<tr><td>Responses</td><td>5</td><td>16</td><td>27</td><td>34</td></tr>
<tr><td>Uploads</td><td>6</td><td>17</td><td>28</td><td>35</td></tr>
<tr><td>Downloads</td><td>7</td><td>18</td><td></td><td>36</td></tr>
<tr><td>New Users</td><td>8</td><td>19</td><td>29</td><td></td></tr>
<tr><td>Calls</td><td>9</td><td>20</td><td></td><td></td></tr>
<tr><td>Time Used</td><td>10</td><td>21</td><td></td><td></td></tr>
<tr><td>Time Idle</td><td>11</td><td>22</td><td></td><td></td></tr>
</table></td></tr>
<tr><td><code>gosub 17</code></td><td><code>gosub 1026</code></td><td>Print the value of <code>st(x)</code> to record <code>x</code> of <code>e.stats</code>. Note that <code>e.stats</code> should be opened first (<code>gosub 30</code>), prior to calling this routine, on LFN 2.</td></tr>
</table>

v3.0	v1.2, v2.0	Purpose
`gosub 28`		Loads a `++` (protocol) file from Plus File drive—if it isn't already loaded—and checks the DOS error status. *Setup:* `a$`: ML or protocol file (minus the `++` prefix) This routine then: 1. displays the module name (`a$`) in the `Area` window of the SysOp screen 2. sets `dr=5` and determines the correct device/drive for the Plus Files system disk 3. checks whether the module requested has already been `load`ed (`ml$=a$`): ◦ If so, the DOS error status (`e%`) is set to `0` to indicate no error, and it `return`s instead of re-`load`ing the file. ◦ otherwise, loads the module via `&,7` and `return`s *Returns:* TODO
`gosub 30`	`gosub 1060`	Opens REL file `e.stats` on Etcetera drive on LFN 2.
`gosub 31`	`none`	Opens REL file `e.access` on Etcetera drive on LFN 2.
`gosub 32`	`gosub 1062`	Opens E-Mail file for desired user. *Setup:* `tt$`: user's handle of the E-Mail file to open (a$) should contain an ",r" or ",w" appropriate for reading or writing.
`gosub 33`	`gosub 1063`	Opens REL file `e.data` on Etcetera drive on LFN 2.
`gosub 34`	`gosub 1064`	Opens SEQ file `e.log (where` is the day code in `am$`) on the Etcetera drive. *Setup:* `a$`: `a` to append, `r` to read, or `w` if doing maintenance that requires creating the file.
`gosub 35`	`gosub 1065`	Opens REL file "u.config" on user files drive.

<table>
<tr><th>v3.0</th><th>v1.2, v2.0</th><th>Purpose</th></tr>
<tr><td>gosub 40</td><td></td><td>Loads sub.editor, and executes at line 60000. This is the entry point for the system editor. Set (mm) according to reason for calling:
<pre>
mm Routine
----
 1 Main Entry Routine (Clear tt$() buffer)
 2 Alt. Entry (Don't Clear, Resume editing)

</pre></td></tr>
<tr><td>gosub 41</td><td></td><td>Loads sub.handles, and executes at line 60000. Set mm according to reason for calling:
<pre>
mm Routine
----
 0 Load u.index and put total
    Number of users in (uh)
 1 Load u.index and check for
    user in (an$). User ID is
    returned in (i), unless not
    found [(i)=0].

</pre></td></tr>
<tr><td>gosub 42</td><td></td><td>Loads sub.protos, and executes at line 60000. <i>Setup:</i> Set mm according to reason for calling:
<table>
<tr><th>mm</th><th>Purpose</th></tr>
<tr><td>0</td><td>Load the file s.m.protos into tt$().
<i>Setup:</i>
b% is set to 1 if in Local mode.
x is set to the total number of protocols in tt$(). (20 max)</td></tr>
<tr><td>1</td><td>Load and display protocol, asks user to select protocol unless in Console Local mode (which defaults to Copier), then loads the protocol.</td></tr>
<tr><td>2</td><td>Load user’s default protocol (found in uh FIXME).</td></tr>
</table></td></tr>
</table>

v3.0	v1.2, v2.0	Purpose

`gosub 43`	`none`	Loads `sub.display`, and executes at line 60000. Set `mm` according to reason for calling:
		`mm  Routine`
		`-------------------------------------`
		` 1  Displays screen used while`
		`    user is online and fills in`
		`    all the user's information.`
		` 2  Wait for Call Screen`
		` 3  Displays screen used while`
		`    user in online, but leaves`
		`    windows blank.`
		` 4  Displays file transfer`
		`    screen where device/drive`
		`    =[dv%(bn+6),dr%(bn+6)]`
		` 5  Displays file transfer`
		`    screen where device/drive`
		`    =[d1%,d2%]`
		`-------------------------------------`

`gosub 50`	`gosub 1490`	Prints `a$` to the daily log, unless in instant mode (`i%=1`). Entering this routine at line 51 ignores `i%`.
`gosub 60`	`gosub 1085`	Writes file `capital reverse P` to device, drive in `dr`, scratches file, then sets `a` to `sgn(e%)` (`0` if `e%=0`, `1` if `e%` is non-zero). This routine is used to test (particularly on floppy-based systems) if there is a free directory entry on the device/drive. It should be called before the creation of any new file.
`gosub 61`	`gosub 1079`	Reads blocks free on device/drive `dr`. This routine should be called and the variable bf checked before creating any file on a device, drive to ensure there is enough space available. Blocks free are returned in the variable (bf).
`gosub 70`		Load and execute an i. file module beginning at line 3000. These are the 'main' modules.
`gosub 72`		Load and execute an i/ 'mini-module' file beginning at line 4000. These are the equivalent of `+.MM.*` files from v1.2.
`gosub 74`	`none`	Load and execute a 'sub.*' module file beginning at line 60000. 'sub' modules are subroutines used to supplement the 'image' file. sub.modem has a subroutine at line 100. (Replaces 2.0’s `im.` files)
`gosub 79`		Loads i.module from device, drive in (dr), then RETURNs. (Lines 70-75 fall thru to lines 76-78 then to this line before returning and executing at the appropriate line).
`gosub 80`		Similar to 24, except uses `p1$` as a reference to currently loaded file, rather than `pr$`.
`gosub 81`		Same as 28, except peculiar difference in approach of checking against `ml$`.

v3.0	v1.2, v2.0	Purpose
gosub 96	gosub 1902	Wait for yes/no hotkey. *Returns:* If Y, then Prints Yes, and a=1. Otherwise No is printed and a=0.
gosub 100	gosub 1013	Load sub.* module in a$ (minus the sub. prefix) from the plus file drive (dr=5), then returns from routine. This routine will also store the filename in (cm$) and output it to the Area window of the SysOp Screen. The "i." and drive designators are automatically added by the sub-routine. If the program (pr$) is already in memory, (e%) is set to 0 to signify no DOS error has occurred, and the sub-routine exits, otherwise this routine exits to line 5 to check the DOS error status. The subroutine filename is added to a “module stack” so that if a sub.* file loads another sub.* file, the previous sub.* file is reloaded on exit. is is the stack depth, and im$() is the module name.
200		System prompt routine. Not to be confused with line 1812 of 1.2’s "im" file which is now line 300, this routine is used for all system prompts. It prints the prompt in (p$), the time, and stack free if in local mode. The routine will check the command stack (uu$), FIXME

v3.0	v1.2, v2.0	Purpose

228		Check for logoff ["O", or "Q" if at Main Prompt (lc=1)] or menu ("?") commands. On 'exit' this routine will goto line 3000 with (mm) set as follows:
		`mm  Action`
		`-------------------------------------------------------`
		` 0  "Init."  Use this as an entry point.`
		` 1  Not a Global (ECS) Command.`
		` 2  Prep. for a prompt display.  (Setup (p$) and`
		`    any pre-prompt text, then RETURN)  This Action`
		`    is called before actually displaying the Time/`
		`    prompt in (p$).`
		` 3  Global (ECS) Command issued.  Clean up & Exit.`
		`    (This Could be a GOSUB or GOTO ECS Command.`
		`    The purpose is to quickly perform a clean-up`
		`    (close files, etc) before proceeding.  In most`
		`    cases, nothing is done.  Exit should be handled`
		`    by issuing a RETURN.`
		`    NOTE: This is also the setting for (mm) that is`
		`    used if the time limit is exceeded.`
		` 4-? *Internal usage by modules*  Not related to`
		`    prompt routine.`
		`-------------------------------------------------------`
		If an ECS command is detected, the routine at line 304 is called. If nothing is entered (<CR>), the local (lc) menu is shown to the user.

`goto 234`		Set `f1=2` for "Immediate logoff" (O!, O%!), otherwise `f1=1`. `cd%` (“carrier drop”) flag. If the 2nd character is a `%` (*i.e.*, `O%` or `O%!`), gosub 302 (load `i.lo`, the logoff module).

<table>
<thead>
<tr><th>v3.0</th><th>v1.2, v2.0</th><th>Purpose</th></tr>
</thead>
<tbody>
<tr><td>goto 250</td><td></td><td>Displays local/level (lc) menu. (See Table)
<table>
<thead><tr><th>lc</th><th>Menu</th></tr></thead>
<tbody>
<tr><td>1</td><td>Main menu</td></tr>
<tr><td>2</td><td>Message Base Menu (SB)</td></tr>
<tr><td>3</td><td>Editor Menu/Help</td></tr>
<tr><td>4</td><td>Local Mode Menu (zz)</td></tr>
<tr><td>5</td><td>File Transfer Menu (UD)</td></tr>
<tr><td>6</td><td>E-Mail Menu (EM)</td></tr>
<tr><td>7</td><td>General Files Menu (GF)</td></tr>
<tr><td>8</td><td>End of Bulletin Menu (SB)</td></tr>
<tr><td>9</td><td>Disk Transfer Menu (UD/UX)</td></tr>
</tbody>
</table>
</td></tr>
<tr><td>gosub 280</td><td></td><td>This routine is called by the prompt routine at line 200 to check for ECS commands.</td></tr>
<tr><td>goto 300</td><td>goto 1812</td><td>Main prompt entry routine. i.main is loaded at line 3000, and executed.</td></tr>
<tr><td>302</td><td></td><td>Loads i.lo file and executes with mm set to 0 (init). Action is dependent on the value of f1:
<table>
<thead><tr><th>f1</th><th>Action</th></tr></thead>
<tbody>
<tr><td>0</td><td>connection established</td></tr>
<tr><td>1</td><td>normal logoff</td></tr>
<tr><td>2</td><td>fast logoff (O! or out of time)</td></tr>
<tr><td>3</td><td>normal entry (when loading i.lo for “Wait For Call” screen)</td></tr>
</tbody>
</table>
</td></tr>
<tr><td>304</td><td></td><td>Reverts memory marker back to 1 (&,28,1), calls line 306 which then issues an &,27 (save) and exits.</td></tr>
<tr><td>306</td><td></td><td>Image 1.2 Emulation Mode. Dimensions variables similar to Image v1.2 [bb$(31), dt$(61), ed$(61), nn$(61), a%(61), c%(61), d%(61), e%(31), f%(61), ac%(31), so%(31)].
“Emulating” 1.2 is not the only use—this routine is helpful to save space and quickly dimension common variables to be used in a program.</td></tr>
<tr><td>gosub 310</td><td>none</td><td>Check for ^?, the command history. If so, goto 315.</td></tr>
<tr><td>gosub 311</td><td>none</td><td>Check for ^ and a digit 0-9. This executes that command history entry.</td></tr>
<tr><td>gosub 315</td><td>none</td><td>Prints up to the last 10 commands (stored in the history stack, hs$(10)) typed via ^?.</td></tr>
</tbody>
</table>

v3.0	v1.2, v2.0	Purpose
gosub 320	FIXME	Update access level of user online. (Called by prompt routine at line 200).
gosub 321	none	If fl$ is not as long as the record in e.access, append the additional flags. This is done when a user previously on Image 1.*x* is upgraded to Image 2.0 or 3.0, since these versions have more user flags per account.
gosub 330	gosub 1096	Outputs a random macro from file e.macros.

goto 999	goto 1603	return jump-point.
		If an on-goto statement needs to exit a subroutine, you can write:
		`1 on a goto 999`
		`999 return`

3.5. sub.* Module Jump Table

Files with a sub. prefix contain routines which are used often, but to save memory and avoid code repetition, are separated from the main im module.

3.6. & Routines

& is the command character which is BASIC's interface to 70 machine language routines of Image BBS.

> This section is undergoing discovery of what some routines are for and how they are used.

3.6.1. &: Image BBS Output

In BASIC, the print keyword (or its abbreviation, ?) displays text on the screen, as in:

```
print"Hello there!"
```

You can substitute & in Image BBS to do the same thing:

```
&"Hello there!"
```

But unlike BASIC, after outputting this text, the cursor remains on the same line—there is no automatic carriage return printed.

It's as if a semicolon (`;`) was used after the `print` statement above—this keeps the text all on one line:

```
print"Hello ";"there!";
```

vs. printing on two lines:

```
print"Hello"
print"there!"
```

To work around this, you can output a *carriage return*, which moves the cursor to the beginning of the next line.

> *Boring Background: CR/LF, CR, LF?*
>
> On other terminals or operating systems, a carriage return moves the cursor to the beginning of the current line.
>
> Sometimes an additional *linefeed* character is needed, which keeps the cursor in the current column, but moves the cursor down a row.
>
> ```
> Hello
>      there!
> ```
>
> This stairstep-looking result is what printing looks like with only a linefeed, when the terminal needed both a carriage return and a linefeed. Probably not the desired result.
>
> However, `print`ing in Commodore BASIC—and our BBS in this example—doesn't need a linefeed. A carriage return is the equivalent of a carriage return plus linefeed.

```
&"Hello there!"+chr$(13)
```

`chr$()` (read it as "character string") is a function that outputs a character supplied in parentheses. A carriage return is `chr$(13)`. So this snippet of code will output `Hello there!`, then move the cursor to the beginning of the next line.

There is also a string variable defined in `im`: `r$=chr$(13)` (short for "return"). This is a handy shortcut. You can now code:

```
&"Hello there!"+r$
```

Using the `r$` variable is actually an easy way to write data containing carriage returns to a SEQ file, as you'll see later.

But there's something even handier available.

3.6.2. Encoded Function Keys

Some special characters are difficult (or impossible) to type in BASIC. Or, they might cause problems while reading disk files. Therefore, they have been encoded using the reverse video letters seen while typing function keys when in Commodore's "quote mode."

This eliminates a few difficulties:

- Entering special characters is made easier—instead of `chr$()` codes, you can just tap a function key in quote mode
- Some more simply written code (and BASIC itself) can truncate data when a `,` is encountered in a SEQ file—this character encoding eliminates that problem

Image BBS converts these special characters from their encoded form to readable characters when you:

- use `RD` to read a SEQ file
- use `SB` to read a post
- Use `WF` to `.G`et, `.E`dit, and `.P`ut a file back.

The `image seq reader` utility used in BASIC also does this.

> A new `&` command, `&,15,2`, does the same thing when passed `an$`, the string to translate.

As it relates to carriage returns, though, we can see in the following table:

Table 3. Image BBS Encoded Function Keys

Key	Quoted	Character	Key	Quoted	Character
`f1`	`E`	`,` comma	`f2`	`I`	`?` question mark
`f3`	`F`	`:` colon	`f4`	`J`	`=` equal sign
`f5`	`G`	`"` quotation mark	`f6`	`K`	`Return` `chr$(13)`
`f7`	`H`	`*` asterisk	`f8`	`L`	`^` up arrow

Key is what you type on the Commodore 64 keyboard.

Quoted is what it looks like in "quote mode."

Character is what the encoded character represents.

3.6.3. `&` By Itself

This is a quick way to output the contents of `a$`.

```
a$="Hello theref6":&
```

This outputs `Hello there` and a carriage return.

We can take the previous example of:

```
&"Hello there!"+r$
```

and simplify it further with:

`&"Hello there!`‍`f6`‍`"`

Because of a quirk in the BASIC interpreter, you must follow a `then` clause with a colon before using the ampersand.

In other words, the following results in a `?syntax  error`:

```
if b then &"hello"
```

This must be used instead:

```
if b then:&"hello"
```

3.7. BASIC Editing Modes

3.7.1. Quote Mode

In the BASIC editor, once the quote mark (`Shift` + `2`, `"`) is typed, color or cursor controls stop changing the cursor color or moving the cursor. Instead, they start displaying reversed characters which stand for the color control, cursor control or symbol you are typing. Once the text inside the quotes is `print`ed, they perform their functions as if typed manually. The `Inst/Del` key is the only cursor control not affected by “quote mode.”

Typing a second `"` exits quote mode, and allows you to use the cursor keys to edit the program line again.

See `&,70` Position Terminal Cursor for how to easily move the terminal cursor within Image BBS, instead of using lots of cursor controls in quote mode.

3.7.2. Insert Mode

Insert mode is similar to quote mode, only for the number of spaces you insert with the `Shift` + `Inst/Del` key, the BASIC editor is in quote mode. Once the space inserted has been typed in, insert mode is exited.

`Return` gets out of quote and insert mode, and adds the current line into the BASIC program.

`Shift` + `Return` gets out of quote and insert mode, but does *not* add the current line into the BASIC program.

3.8. Outputting Strings

3.8.1. String Literals

3.8.2. String Variables

3.8.3. Concatenation

3.8.4. Word-Wrap

A new feature of the text output routine is *word-wrap*. This is where a word which would extend past the right hand margin is moved to the following line instead.

The position to wrap words is usually 40 columns on the console, whatever the remote user's screen width is, and 80 columns on the printer, if one is attached.

Setting the variable `lp=1` and outputting text with `&"…"` will enable word-wrap.

Setting `lp=0` disables word-wrap, so words continue past the right margin.

3.8.5. Set Margins

The left and right margins can now be indented by up to 15 characters using two new MCI commands:

- `£m<x` (set left margin). This command causes every carriage return issued by Image BBS to be followed by *x* spaces, which indents text *x* spaces. The values for *x* are `0` (to disable word-wrap), or `1-9`, and `j-o` (10-15).
- `£m>x` (set right margin). This command causes the system to word wrap as if the user's screen width was *x* characters less than it really is. It indents the text from the right side.

The use of `£m<x` and `£m>x` together allows you to make "block indents" of text that appear correct regardless of the user's screen width.

3.8.6. MCI Commands

TODO

Outputting a string which itself contains MCI commands or MCI string variables will not work as expected, *e.g.*:

```
c$="Hi":c%=3:z$="£$c £#3£#0£%c":&"£$z"
```

does not output `z$` (`£$c` outputs `c$`, `£#3` sets 3 leading characters, `£#0` sets the fill character to `0`, and `£%c` displays the value of `c%`). It outputs a literal

```
£$c £#3£#0£%c
```

Instead, do this:

```
c$="Hi":c%=3:z$="£$c £\#3£# £%c":&z$
```

This outputs the expected

```
Hi 003
```

3.9. Outputting Numbers

3.9.1. Integers

3.9.2. Floating Point

Floating point values are not integers—the value could be fractional, and the decimal point could be in any position, hence the term “floating point.”

`&str$(h)`

Since `&h` isn’t supported to output the value of `h`, it must be converted to a string with the `str$()` function.

`&str$(h)` is still useful if the number of digits output is greater than 5, the limit of the MCI numeric output routine.

`&str$(h)` worked under Image 1.2; it works here too, but there is a new MCI command in Image 3.0 to output single-letter floating point variables:

`&"£!h"`

Output the value of `h`.

To output a two-character variable name (*e.g.*, `xx`, `c1`), you have some choices:

- assign the two-character variable name to a single letter variable name, *e.g.*:

```
x=xx:&"Blocks free: £!x"
```

- output the two-character variable using the `str$()` function:

  `&"Number of days:"+str$(xx)`

  Output the value of `xx`.

Chapter 4. `&` Parameters Explained

`&` commands from this point on have additional parameter(s) called by at least one number, followed by optional parameter(s). These variables are denoted in *italics*.

`&,call`

This means that `&,` is required, but the value of *call* varies. Substitute the appropriate number in its place.

Example

`&,40`: perform garbage collection

`&,call[,optional]`

This means that, again, `&,call` is required. The value of *call* varies. However, the parameter *[,optional]* is not required. If it is supplied, there needs to be a `,` and the appropriate parameter substituted in its place.

Example

`&,9,1`: display `a$` in the 16-character programmable window

There are various sections which most commands use. They outline BASIC setup (variables or `poke`s) which need to be done before the `&` call can be used.

BASIC Setup:

Any `poke`s or BASIC variables which are used by the `&` command are listed here.

Parameters:

Any extra information given after `&,x`, like strings or numbers, are listed here.

Returns:

Strings, numbers or arrays returned when the `&` command is finished are listed here.

Examples:

Examples of setup, the `&` call being used, and some typical results are listed here.

4.1. `&,1` Input Line

`&,1` accepts input from the user at a prompt. It handles features including word wrap, MCI access, line length, and the ability to type graphics characters.

BASIC Setup:

`poke 53252,line_length` limits the length of the input to *line_length* characters, 1-79.

`p$`: text of the prompt shown before `:` and input is accepted.

`w$`: the default response to a prompt. When using edit mode, if only `Return` is typed at a prompt, `an$`

is set to a null string (`an$=""`). The module can check for this and not update the original string. An example is given below.

`pl=0`: allow both lowercase and uppercase.

`pl=1`: convert lowercase input to uppercase.

Parameters:

`&,1,editor[,password]`

Not all of these parameters are currently understood.

`editor`: editor flags:

Each bit controls a separate function of the input routine. Bits may be combined together to perform multiple functions.

Binary	Decimal	Purpose (if set)
`%00000001`	1	disable typing graphics characters
`%00000010`	2	`.` or `/` on column one exits input
`%00000100`	4	disable prompt (`p$`)
`%00001000`	8	disable typing `£` (the MCI command character)
`%00010000`	16	enable word-wrap
`%00100000`	32	enable edit mode
`%01000000`	64	ignore time remaining
`%10000000`	128	disable typing `Delete` on column one to exit input

Returns:

Variable	Purpose
`an$`	1

`peek()`	Purpose
`$d006/53254`	0: normal 1: `Delete`, `.` or `/` typed on column 1

Explanations:

TODO: demonstrations of each mode in `i.test calls`

Edit mode

The prompt (`p$`) is displayed, and the default response (`w$`) is displayed. Then, the prompt (`p$`) is displayed again, and one of three choices can be made:

- `Return` accepts the default (`w$`)
- a new string can be typed
- the current string can be edited using `Ctrl` key shortcuts

`password`: password flags:

Binary	Decimal	Purpose
`%00000001`	1	password mask enabled for output [uses mask character in `peek(17138)`]
`%00000010`	2	no output

Returns:

`an$`: text typed at the prompt.

Examples:

TODO: write examples for each option.

Example 1:

```
poke 53252,10:p$="Name":&,1
```

`poke 53252,10`: Set the input length to 10 characters.

`p$="Name"`: Set the prompt to `Name:`.

`&,1`: Allow user input, and return string in `an$`.

Example 2:

```
poke 53252,20:w$=na$:p$="Handle":&,1,32:if an$<>"" then na$=an$
```

`poke 53252,20`: Set the input length to 20 characters.

`w$=na$`: Assign the user's handle (`na$`) to `w$` (the prompt's default).

`p$="Handle"`: Set the prompt to `Handle:`.

`&,1,32`: Setting bit 5 (a value of 32) enables edit mode. This displays the prompt, the original text (`w$`), and re-displays the prompt. Editing control keys can be used to change the input. Editing mode looks like this:

```
Handle: PINACOLADA

```

```
Handle: _
```

The string typed in response to a prompt is returned in `an$`.

`if an$<>"" then na$=an$`: Just typing `Return` (to accept the default, `w$`) sets `an$` to a null string. If something else was typed (`an$<>""`), assign `na$` to what was typed (`an$`).

Example 3:

```
poke 53252,30:p$="New Prompt":&,1,9:if an$<>"" then po$=an$
```

`poke 53252,30`: Set the input length to 30 characters.

`p$="New Prompt"`: Set the prompt to `New Prompt:`.

`&,1,9`: Setting bits 3 and 0 (a value of 8 + 1) allow MCI and graphics characters to be entered.

4.2. `&,2` Disk File Input

Input data from an open disk file. This routine inputs a maximum of 80 characters into `a$`.

Parameters:

`&,2,lfn[,bytes]`

`lfn`: logical file number.

> *Logical File Numbers*
>
> A logical file number (LFN) relates the `open`, `close`, `cmd`, `get#`, `input#` and `print#` statements to each other.
>
> LFNs help distinguish multiple files from each other. They associate the commands being used with the filename opened which uses the same LFN.
>
> The LFN is the first number in an `open` statement, *e.g.* `open 2,10,2`.
>
> The LFN value ranges from 1 to 255. LFN 15 is usually reserved for the DOS command channel.
>
> The only restriction is that you can’t re-`open` a LFN that is already `open`, or you get the error `?file open  error`.

`bytes`: number of bytes to get from file (1-80). Carriage returns are ignored.

Returns:

`a$`: bytes from file

Example:

s.test file

```
data ①
----+----+----+----+----+----+ ②
```

① a regular string

② a 30-character string, used to demonstrate `&,2,2,25`

i.read test file

```
3000 dr=1:a$="s.test file,s,r":gosub 4 ①
3002 if e% then:&:goto 3100 ②
3004 &,2,2:& ③
3006 &,2,2,25:& ④
3100 close 2:goto 300 ⑤
```

① `dr=1`: set the Image drive to `1` (the System drive). `a$="s.test file,s,r"`: set the System disk filename prefix to `s.`, the main filename to `test file`, specifies `s` for a SEQuential file, and `r` for reading the file. `gosub 4`: open `a$` for the device and drive assigned to the System drive.

② If there is a DOS error (*e.g.* `file not found`), this line intercepts it. `e%`: the DOS error number. `if e%` implies `if e%<>0` (if `e%` does not equal zero; *i.e.*, a non-zero result means an error occurred). `a$`: the DOS error string. `&`: display the DOS error string in `a$`. `goto 3100`: close the disk channel instead of getting data from a non-existant file.

③ `&,2,2`: using logical file #**2**, get a line of data from the disk. The data is returned in `a$`. `&`: output `a$`.

④ `&,2,2,25`: using logical file #2, get a line of data from the disk—but stop at **25** characters, instead of getting the entire line. The data is returned in `a$`. `&`: output `a$`.

⑤ `close 2`: close disk file. `goto 300`: go to main prompt.

Example 2. BASIC Pitfall

Using `input#2,a$` when the disk file contains a string `hello,there` returns only `hello` in `a$`. When used with `input#`, `,` is a delimiter which truncates (cuts off) the data after the `,`.

You can prefix the string with `"` on disk to get around that. But most likely, you want to read `hello` and `there` into two separate variables. `input#2,a$,b$` does that, resulting in `a$="hello"` and `b$="there"`.

4.3. `&,3` Read File from Disk

Read a file from disk. An optional *speed* parameter can be specified for reading movie files, which adds an appropriate slowdown based on the value.

Parameters:

`&,3,lfn[,speed]`

`lfn`: logical file number

`,speed`: speed (`1`-`20` for movie file read. `1`=faster, `20`=slower)

4.4. `&,4` Get Byte from Modem

This returns the character received from the modem in `peek(780)`. This routine does no ASCII translation, and no high bit stripping; it gets the character directly from the RS232 routines.

Returns:

`x=peek(780)` reads the character from the RS232 routines. If no character is received from the modem, `peek(780)=0`.

4.5. `&,5` Get Version

Get the version information embedded in the BBS ML.

Returns:

`lp`: major/minor (1.3)

`a%`: revision (1)

`a$`: date (`"12/29/91 1:18p"`)

4.6. `&,6` Password Input

Sets the input length to 14, and displays a definable mask character rather than the actual characters typed. The text typed is returned in plain (non-masked) text.

Parameters:

`poke 17138,mask`: display *mask* character instead of the user’s input

You could do `poke 17138,asc("X")` to set the mask character to `X`.

Returns:

`an$`: password in plain text

Example:

im

```
263 &"Password: ":&,6:if an$<>ep$ then:&"Incorrect Password."
```

`&"Password: "`: Display the prompt `Password: `.

`&,6`: Allow password entry, displaying the mask character instead of the text actually typed.

`if an$<>ep$ then:&"Incorrect Password."`: if `an$` (the entered text) is not equal to (`<>`) `ep$` (a password set on an Extended Command Set command), display the message.

4.7. `&,7` Load File

Loads a module into memory.

Parameters:

`a$`: the drive number and filename (*e.g.*, `"0:i.module"`)

Syntax:

`&,7,device[,segment]`

Segments

Segments are pre-defined addresses that a module will load to, regardless of the file's first two bytes which define its load address.

Not all segments are currently defined.

Segment	Purpose
2	Protocols or blocks of ML
7	Print mode table
8	Lightbar text
9	Alarm table
10	ASCII-to-Commodore translation table
11	Commodore-to-ASCII translation table
12	Print mode table

im:

```
29 ml$="++ "+a$:a$=dr$+ml$:&,7,dv%,2:goto 5
```

This line loads an ML protocol (`++` file).

```
ml$="++ "+a$
```

Assign `ml$` the `++ ` prefix for error reporting purposes.

```
a$=dr$+ml$
```

Concatenate `dr$` (the current drive prefix) and `ml$` (discussed above) into `a$`.

```
&,7,dv%,2
```

Do a module load using device `dv%` into segment 2 (the protocol address, `49152` or `$c000`).

```
goto 5
```

Check for a DOS error.

4.8. `&,8` Disk Directory

Display either:

- an entire disk directory at once, from the directory header to the `blocks free.` message
- a line of information at a time (calling it multiple times will get the directory header, each file's block count, filename, filetype, splat and lock status, and the `blocks free.` count)

Parameters:

`&,8,lfn,flag`

`lfn`: logical file #

`flag`: [`0`=entire directory | `1`=single line]

Returns:

`flag=0`: Displays entire directory

`flag=1`: `a$`: single line of disk directory information

Display Entire s. Disk Directory

```
3000 dr=1:gosub 3:open 2,dv%,0,"$"+dr$+"*":get#2,a$,a$ ①
3002 &,8,2,0:close 2:goto 300 ②
```

① `dr=1:gosub 3`: get device of `s.` disk. `open 2,dv%,0,"$"+dr$+"*"`: open the directory as a file. The secondary address must be `0` to instruct the drive to return the disk directory as a BASIC-formatted series of lines, displayable by this routine. `$0:*`: Use the wildcard pattern `*` (all files).

get#2,a$,a$: discard the load address information.

② &,8,2,0: Use lfn#**2** to get the entire disk directory (0). close 2: close lfn#2. goto 300: go to main prompt.

4.9. &,9 Bottom Variable

Output variables to 16-character status window.

Parameters:

&,9[,x]: *x*=variable number to display, 0-49:

Variable	Variable	Variable	Variable	Variable
0 an$	10 tt$	20 nl	30 a%	40 f1$
1 a$	11 na$	21 ul	31 b$	41 f2$
2 b$	12 rn$	22 qe	32 dv%	42 f3$
3 tr$	13 ph$	23 rq	33 dr$	43 f4$
4 d1$	14 ak$	24 ac%	34 c1$	44 f5$
5 d2$	15 lp	25 ef	35 c2$	45 f6$
6 d3$	16 pl	26 lf	36 co$	46 f7$
7 d4$	17 rc	27 w$	37 ch$	47 f8$
8 d5$	18 sh	28 p$	38 kp%	48 mp$
9 ld$	19 mw	29 tr%	39 c3$	49 mn%

Examples:

&,9[,0]: output an$ to status window

&,9,1: print a$ to status window

&,9,2: print b$ to status window

&,9,3: output tr$ to status window

&,9,4: output d1$ to status window

4.10. &,10 Terminal Mode

C= + Ctrl leaves terminal mode

4.11. &,11 Clear Array

Clear array #*x*.

Table 4. Array Numbers

Number	Array	Purpose
0	`tt$()`	Text editor: lines entered
1	`bb$()`	
2	`dt$()`	
3	`ed$()`	
4	`nn$()`	
5	`a%()`	
6	`c%()`	
7	`d%()`	
8	`e%()`	
9	`f%()`	
10	`ac%()`	Access levels
11	`so%()`	Subops

Example:

`&,11`

Clear `tt$()` array.

4.12. &,12 New User

Non-abortable file read.

4.13. &,13 arbit

A function reserved to arbitrate port use in multi-port Lt.Kernal hard drive setups.

ℹ This is currently being researched.

4.14. `&,14` Dump Array

Write array elements to an already-open file, using logical file #2.

`&,14,array`

Output from 1 to however many elements were `dim`ensioned for *array*.

`&,14,array,end`

Output elements of *array* from 1-*end*.

See Array Numbers for the arrays which correspond to *array*.

4.15. `&,15` Convert `an$`

This group of functions perform various conversions on the string contained in `an$`.

Related string conversion functions can be found at `&,65` convert.

4.15.1. `&,15` Convert Date

`an$=d1$:&,15:&an$` → displays verbose date

1. `an$=d1$`: Put current 11-digit date (`d1$`, *e.g.* `60429218427`) into `an$`
2. `&,15`: Convert 11-digit date to a long date string, *e.g.* `Thu Apr 29, 2021  4:29 P` and assign that to `an$`
3. `&an$`: Output `an$`

Image BBS Date Format

Image BBS uses an 11-digit string to represent a time and date. The format is `w yr mo dt hr mi`.

Extra spaces between the numbers have been added for ease of reading, but are not used in the actual string.

Table 5. Image BBS Date Format

Position	Abbreviation	Purpose	Range
1	`w`	weekday	`1`=Sun…`7`=Sat
2-3	`yr`	year	`00`…`99` (the year within the century, `20xx`, is displayed using `&,15`)

Position	Abbreviation	Purpose	Range
4-5	`mo`	month	`01`…`12`
6-7	`dt`	date	`01`…`31`
8-9	`hr`	hour	`00`…`11` (12:00 AM, midnight—11:00 AM, 1 hour until noon) `80`…`92` (12:00 PM, noon—11:00 PM, 1 hour until midnight)
10-11	`mi`	minute	`00`…`59`

The current time and date is stored in the string `d1$`, which is continually updated by the ML. Here is a sample definition of `d1$`:

```
"22105178944"
```

Let's break down how the string is encoded.

Table 6. Image BBS Date Decoding

Position	Value	Purpose	Meaning
1	`2`	weekday	`Mon`
2-3	`21`	year	`2021`
4-5	`05`	month	`May`
6-7	`17`	date	`17`
8-9	`89`	hour	` 9:00 PM` (`9`=hour, plus `80`=PM)
10-11	`44`	minute	`44`

As the table above shows, this string stands for `Mon May 17, 2021  9:44 PM`.

You can also output `Ctrl` + `d` or `chr$(4)` in an `&` statement to convert an 11-digit date/time string to a long date/time string. Both these statements output the same string as above:

- `a$="22105178944":&"Ctrl□+□d"+a$`
- `a$="22105178944":&chr$(4)+a$`

Outputting the date and time this way also outputs the user's time zone.

4.15.2. `&,15,1` Title Case

Changes an all uppercase string to mixed case.

`an$="THE CHIEF":&,15,1:&an$` → `The Chief`

4.15.3. `&,15,2` Decode Function Keys

Decodes quoted function key characters into readable equivalents.

```
i.t
   an$="host f3 port":&,15,2:&"£v7 f6 " → host:port
```

4.15.4. `&,15,3` and `&,15,4`

These point to, and are the same as, `&,15,2`.

4.15.5. `&,15,5` newdate

Some sort of hour (?) conversion.

Syntax:

`an$="wyymmddhhmm":&,15,5:&" Ctrl □+□ d "+an$`

ℹ This function is currently being researched.

4.15.6. `&,15,6` Scan String

Scan `an$` for the first occurrence of a specified character. You can specify the character to scan for in one of two ways:

- *x*, the PETSCII value of the character
- use the `asc("x")` function, which returns the ASCII (or PETSCII) value of character *x*

If the specified character is found in `an$`, split it into two strings:

- `an$` now ends before the specified character was found
- `a$` begins after the specified character was found to the end of the string

Split on space, two ways

1. `an$="Hello world":&,15,6,32`

This splits `"Hello world"` at `chr$(32)` (`Space`), resulting in `a$="Hello"` and `an$="world"`.

2. `an$="Hello world":&,15,6,asc(" ")`

This splits `"Hello world"` at the **ASCII value of** `Space` (32), resulting in `a$="Hello"` and `an$="world"`.

After the split, the two strings look like this:

`a$="Hello"`	`chr$(32)`	`an$="world"`

If the specified character is not found in `an$`, `a$=""`, a null string.

im

`312 &,15,6,140:uu$=an$:an$=a$`

`140`=`f8`, Image `^`

```
352 &,15,6,133:d2%=val(an$):d1%=a:dr=.:dv%(.)=d1%:dr%(.)=d2%
```

`133`=`f1`, Image `,`

4.16. `&,16` sys49152

Perform `sys 49152`. Usually this is used in a file transfer protocol for performing a file copy, upload, or download.

Parameters:

`&,16`[,*sub-function*]

Following `&,16` with a *sub-function* number (*e.g.*, `&,16,2`) calls a sub-function of the module through a *jump table*.

Details: Jump Tables

A *jump table* is used to give assembly language routines stable entry point addresses that don’t change even if routines pointed to change in size. This is done by maintaining a list of addresses to be `jmp`ed to in 6510 assembly code.

In other words: when *not* using a jump table, changing a routine’s size shifts subsequent inline routine entry points around by the number of bytes added or subtracted by the modification. You probably don’t want to search for all the BASIC `sys` addresses referencing the changed entry points throughout your code.

Instead, just change the address that the jump table entry `jmp`s to, and you can keep the BASIC `sys` address that calls the routine the same (stable).

As an example, each of these instructions in a fictitious protocol assembly-language jump table starting at `$c000` take 3 bytes:

`c000: jmp $c009` ; `sys 49152` sub-function 0 (`&,16`)

`c003: jmp $c0a5` ; `sys 49155` sub-function 1 (`&,16,1`, also the equivalent of `&,17`)

`c006: jmp $c147` ; `sys 49158` sub-function 2 (`&,16,2`)

`c009:` <*first byte of first routine*>

Refer to Protocols for more information and listings of jump tables.

4.17. `&,17` sys49155

Perform `sys 49155` as shown above.

4.18. `&,18` Set Screen Mode

This command turns the screen mask on (enabling split screen mode) or off (enabling full-screen mode). The bottom status line (showing the date and time, status indicators, and the time remaining for a user’s call) is always present, regardless of mode.

Parameters:

`&,18,0`: Turn the screen mask off, enabling full screen mode (24 lines for viewing caller activity).

`&,18,1`: Turn the screen mask on, enabling split screen mode (16 lines for viewing caller activity). The 9 lines of the screen mask show:

- the lightbar interface
- system, caller, protocol or network transfer information depending on the BBS’s mode
- the modem I/O windows, `M=` free memory, and `L=` BASIC line number currently executing

4.19. &,19 Get Version

This function did something different in Image 1.2, but was removed. It points to &,5: Get Version to maintain the numbering of the & calls.

4.20. &,20 Read from Interface Table

Reads a byte from the interface table. This is meant to possibly eventually replace peek()ing memory locations. While the functionality is there, it is limited, but can be expanded in the future.

Parameters:

`&,20,index,command`

index: position in table (see table)

command: [0=put in a% | 1=return in accumulator, peek(780)]

Table 7. Interface Table Addresses

Index	peek()	Hexadecimal	Purpose
0	53252	$d004	Input line length

For now, refer to Pokes and Memory Locations to see the list of pokes you may use.

4.21. &,21 Write to Interface Table

Writes a byte to the interface table. This is meant to possibly eventually replace pokeing memory locations. While the functionality is there, it is limited, but should be expanded in the future.

Parameters:

index: see Interface Table Addresses for more information

value: the value you would normally poke into a memory location

Example:

&,21,0,20 Set input line length to 20 characters.

4.22. `&,22` Wait *x* Tenths of a Second

This waits for any interval from .1 second to 25.5 seconds, in 1/10-second steps.

Parameters:

`x`= 1-255

Example:

```
&,22,10  ①
&,22,200 ②
```

① Wait 1 second (10 10^{ths} of a second)

② Wait 20 seconds (20 $10/10^{\text{ths}}$ of a second)

4.23. `&,23` Get Character from Modem

```
3000 &,23:c=peek(780):if c<>32 then 3000 ①
```

① Get character from modem. Save in `c`. Loop until the caller hits `Space` [`chr$(32)`].

> 💡 `&,23` doesn’t stop and wait for input, unlike the `£g1` MCI command. If no character is received from the connected user, `peek(780)=0`.

4.24. `&,24` xchrout1

This is an output character routine that should be used when writing ML routines which need to output a character to the user.

4.25. `&,25` Sound

Produce 4 separate sounds, optionally repeating the sound a specified number of times.

Parameters:

`&,25,sound[,repeat]`

sound=

```
&,25,0  beep
&,25,1  ding
```

```
&,25,2  higher pitched ding
&,25,3  gong sound from CCGMS (a terminal program)
```

`repeat=` Number of times to repeat: [`0`: Stop repeat | `1-254`: Repeat count | `255`: infinite]

4.26. `&,26` ecschk

ℹ This is currently being researched.

4.27. `&,27` Save Variable Pointers

This saves pointers that tell BASIC where variables and arrays start and end. When modules introduce arrays not already defined in `im`, the variable pointers can be restored with `&,28`. This erases unnecessary variables after they're done being used.

FIXME

4.27.1. Creating New Arrays

If you define any new arrays in a module, be sure not to consume unnecessary memory after you end the module. You can do this by using the `&,27` (array pointer save) and `&,28` (array pointer restore) calls.

> Image 1.2 had just one level of variable pointer save and restore. Image 1.3 and above adds multiple levels of save and restore with an additional parameter.

```
3000 &,27,2 ①
3002 dim u%(10,20) ②
...
3010 &,28,2:goto 300 ③
```

① save current variable pointers, and create variable pointer level 2

② create new array

③ `&,28,2`: restore level 2 array pointers (this frees up memory used by the array but preserves level 1 system variables still needed by the BBS). `goto 300`: go to main prompt.

TODO: I would like a diagram of array pointers, creating new arrays, restoring old pointers here.

💡 The main prompt restores level 1 array pointers and already does a `&,28,1` there.

If you substitute `&,27,1` for `&,27,2` and `&,28,1` for `&,28,2` in the above code, line 300 will redo `&,28,1`. This causes a `?redim□d array  error in 306` (*i.e.*, redimensioned array error; an array can't be `dim`ensioned twice).

4.28. `&,28` Restore Variable Pointers

FIXME

4.29. `&,29` usevar

Get contents of a variable. This is the routine to call to read the value of a variable from ML.

Prerequisite:

`ldx` *variable_number* ; variable to access.

(Refer to variable table FIXME.)

Returns:

`$61`: Start of buffer holding the variable contents.

4.30. `&,30` putvar

Assign a value to a variable. This stores the contents of the buffer at `$61` into a variable.

Prerequisite:

`ldx` *variable_number* ; variable to access.

(Refer to variable table FIXME.)

Returns:

`$61`: Start of buffer holding the variable contents.

4.31. `&,31` zero

This stores the floating point equivalent of `0` in the buffer starting at `$61`.

4.32. `&,32` minusone

This stores the floating point equivalent of `-1` in the buffer starting at `$61`.

4.33. `&,33` getarr

Get descriptor (length and pointer) for an element of `tt$(x)`.

Prerequisite:

`ldx` *element* ; element to access.

Returns: `$61`: Start of buffer holding the descriptor.

4.34. `&,34` putarr

4.35. `&,35` getln

4.36. `&,36` putln

4.37. `&,37` trapon

Enable error-trap routine. BASIC run-time errors will be caught, and redirected to the BBS error handler at line 2000.

4.38. `&,38` trapoff

Disable error-trap routine. BASIC run-time errors will not be caught, and will crash like in regular BASIC, halting the program and putting you back at the `ready.` prompt.

4.39. `&,39` prtln

Prints the array element `tt$()` contained in the `.x` register.

4.40. &,40 forcegc

Perform a garbage collection (freeing RAM by erasing unused strings). While the garbage collection is being performed, `G` shows in the status indicator area on the bottom status line.

`FG` also performs garbage collection if you are in pseudo-local mode.

4.41. &,41 setbaud

This command changes the *bits per second* (BPS) rate.

Do not change the BPS rate while someone is online. This only changes the rate of Image BBS transmitting data; the modem cannot match speeds except while offline.

Parameters:

Parameter	BPS rate
`&,41,0`	300
`&,41,1`	*not used*
`&,41,2`	1200
`&,41,3`	2400
`&,41,4`	4800
`&,41,5`	9600
`&,41,6`	19200
`&,41,7`	38400

To utilize speeds higher than 2400 BPS, you must have a SwiftLink, Turbo232 or compatible high-speed RS232 cartridge connected to the expansion/cartridge port of your Commodore 64.

4.42. &,42 ECS Commands

This group of commands are used for interfacing the Extended Command Set (ECS) with BASIC. There are sub-commands to:

- load and save the command set

- search for and update individual commands
- `goto` or `gosub` line numbers in `im`, or modules

4.42.1. `&,42` Check for ECS Command

This checks whether the command passed in `an$` is a valid ECS command.

Example:

im

```
226 f4=.:a%=zz:b%=2^ac%:&,42:if a% then ef$=b$:ep$=a$:ec=a%:ec%=b%:goto 261
```

BASIC Setup:

`an$`: command the user typed

`a%`: Local Mode flag (`zz`)

`b%`: access level

Returns:

`a%`: [`0`: not found in ECS table | `n`: ECS command #`n`]

`a$`: password

`b$`: ECS flags

`b%`: credits to use command

4.42.2. `&,42,1` Goto Line in ECS Command

This will `goto` a particular line in `im` contained in the ECS command, if its `goto/gosub` flag is set to `goto`. In this respect, it is similar to `&,66`: `calculated GOTO`.

Parameter:

`a%`: line number to `goto`

Example:

im

```
268 a%=asc(ef$+nl$)+256*asc(mid$(ef$,2,1)+nl$):&,42,1
```

From an ECS flags string `ef$`, `a%` holds the line # to `goto`, using `&,42,1`.

4.42.3. `&,42,2` Get ECS Definitions From RAM

Example:

i/IM.ecs

```
4004 &,42,2:n=a%:goto 4034
```

Returns:

`a%`: number of ECS definitions in memory

4.42.4. `&,42,3` Put ECS Definition Into RAM

Add/replace the ECS definition in `tt$(n)` to the list currently in memory.

Parameter:

`tt$(n)`: command definition

`&,42,3,number`: the command number in the ECS to add/replace

Examples:

i/IM.ecs

```
4010 tt$(n+1)=chr$(0):&,42,3,n+1:return
```

`n`: the current count of ECS commands

Assign an empty command [`chr$(0)` is a null byte] to the next command [`tt$(n+1)`].

`&,42,3,n+1`: Add the empty command in [`tt$(n+1)`] to the ECS.

4.42.5. `&,42,4` Load ECS Definitions from Disk

Load ECS definitions from a disk file.

Parameters:

`a$`: filename

`dv%`: device #

Example:

im

```
3106 a$=dr$+"e.ecs.main":&,42,4
```

Load the ECS definitions in `e.ecs.main` from disk.

4.42.6. `&,42,5` Save ECS Definitions to Disk

Save ECS definitions to a disk file.

Parameters:

`a$`: filename

`dv%`: device #

BASIC Setup:

> 💡 Line 4016 is the root example, lines 4010 and 4018 are provided for context.

i.IM/ecs:

```
4010 tt$(n+1)=chr$(0):&,42,3,n+1:return ①

4016 &"Save To Disk{f6:2}":gosub 4010:gosub 4018:gosub 19:a$=dr$+a$:&,42,5:tz=0:return
②

4018 dr=3:a$="e.ecs.main":return ③
```

① Set end of ECS command list.

② `gosub 4010`: see callout 1. `gosub 4018`: see callout 3. `gosub 19`: scratch existing `e.ecs.main` file. `&,42,5`: Save ECS definitions to disk. `tz=0`: Clear “file modified” flag.

③ assign `dr=3` (Image drive) to the Etcetera disk, and `a$="e.ecs.main"`, the ECS filename.

4.43. `&,43` chatchk

Checks for presence of the `Cht` left check mark.

4.44. `&,44` trace

Checks for presence of the `Trc` left check mark.

4.45. `&,45` prtvar

Prints a variable with MCI enabled.

4.46. `&,46` prtvar0

Prints a variable with MCI disabled.

4.47. `&,47` carchk

Checks for the presence of a Carrier Detect signal.

Returns:

`0`: carrier present, or local mode

`1`: carrier dropped

`2`: timeout

4.48. `&,48` getkbd

Check console keyboard for a keypress. This can also be also used from BASIC.

Returns:

`peek(198)`: Character typed

> 💡 `&,48` does not stop and wait for input, unlike the `£g1` MCI command. If no character is typed on the keyboard, `peek(198)=0`.

FIXME: verify this

4.49. `&,49` getmod

Gets a character from the modem, with ASCII translation.

4.50. `&,50` outscn

Output a character to the BBS console.

4.51. `&,51` outmod

Outputs character in accumulator [`peek(780)`] to modem.

4.52. `&,52` Lightbar Interface

This is how to read the status of lightbar checkmarks, change checkmarks' status, and move the "lit" portion to a specific location.

Parameters:

`&,52,position,option`

`&,52,$hexadecimal,option` is allowed.

Example: Turn off `Trc` left: `&,52,$18,0` (`&,52,24,0` decimal).

`option=0`: clear checkmark at *position*

`option=1`: set checkmark at *position*

`option=2`: toggle checkmark at *position*

`option=3`: read status of *position*, return in `a%` (`0`=off, `1`=on)

`option=4`: move lightbar to *position*

Option 5 does the same thing in ML as option 4 does in BASIC.

4.53. `&,53` Logoff

- Resets various flags:
  - chat page counter
  - sound repeat counter
- clears status line indicators

Example:

im:

```
3074 &,53
```

4.54. &,54 Text Editor Interface

The text editor is called with the `&,54,mode` command. The editor can be entered in different ways, depending on the value of *mode*.

Line 60100 of `sub.editor` uses `&,54,a`. The values of *a* and the purpose for entering the editor are as follows:

Table 8. Text editor calls

Line	Variable	Purpose
`60086`	`a=0`	Normal entry, empty buffer
`60110`	`a=1`	Buffer not cleared
`60108`	`a=2`	Extended editor command

When `a` is equal to 0, this is the "normal" entry point into the editor with an empty buffer.

When `a` is equal to 1, entry into the editor does not clear the buffer.

When `a` is equal to 2, entry into the editor is the point that would be used in an extended command if the command that was typed was not a recognized command.

How to use the editor in your BASIC programs is described later in “The Editor.”

4.55. &,55 output

4.56. &,56 chatmode

4.57. &,57 relread

Reads records from an open RELative file until the character `^` is encountered.

Parameters:

`&,57,lfn`

`lfn`: logical file number

4.58. &,58 setalarm

Parameters:

`&,58,hour,minute`

`hour`: hour

`minute`: minute

4.59. &,59 farerr

Cause a specified BASIC error to happen.

Parameter:

`&,59,1,error`

Example:

```
&,59,1,14
```

Cause error `14`, `?illegal quantity  error`.

Table 9. Error Numbers

Number	Error	Number	Error
1	TOO MANY FILES	16	OUT OF MEMORY
2	FILE OPEN	17	UNDEF’D STATEMENT
3	FILE NOT OPEN	18	BAD SUBSCRIPT
4	FILE NOT FOUND	19	REDIM’D ARRAY
5	DEVICE NOT PRESENT	20	DIVISION BY ZERO
6	NOT INPUT FILE	21	ILLEGAL DIRECT
7	NOT OUTPUT FILE	22	TYPE MISMATCH
8	MISSING FILE NAME	23	STRING TOO LONG
9	ILLEGAL DEVICE NUMBER	24	FILE DATA
10	NEXT WITHOUT FOR	25	FORMULA TOO COMPLEX
11	SYNTAX	26	CAN’T CONTINUE
12	RETURN WITHOUT GOSUB	27	UNDEF’D FUNCTION
13	OUT OF DATA	28	VERIFY

Number	Error	Number	Error
14	ILLEGAL QUANTITY	29	LOAD
15	OVERFLOW	30	BREAK

4.60. `&,60` Structures

Structures (or *structs* for short) allow you to access and manipulate the memory used by arrays at machine language speeds.

See the Structures chapter for more information.

4.61. `&,61` poscrsr

Move the terminal cursor to a specified column and row on the screen.

Parameters:

`&,61,column,row`

> ℹ `0,0` is the top left corner of the screen.

im

```
4004 &,38:&,61,.,8:print"&,37:goto300:":end
```

`&,61,0,8`: position the cursor at column 0, row 9. `print` recovery information if the BBS crashes.

4.62. `&,62` Set Time

Set BBS clock.

Parameters:

`&,62,` *hour*, *minute*

`hour`: hour

`minute`: minute

Example:

im:

```
3182 &,62,h,m
```

4.63. &,63 inline1

4.64. &,64 convstr

4.65. &,65 convert

Another group of related string conversion functions, there are sub-functions to convert:

- names (?)
- disk data
- special characters to Image-encoded function keys, and
- check special characters.

Related string conversion functions can be found at &,15 Convert an$.

4.66. &,66 Calculated goto

goto line number held in a%. If you have 13 4-digit goto targets on a line, this can save a fair amount of RAM (and BASIC interpretation time).

BASIC Setup:

a%: line number to goto

Instead of writing this (which uses 73 bytes):

Spaces between line number targets have been added for clarity's sake.

Example 3. Calculated GOTO

```
on a% goto 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 4100,
4200, 4300
```

The same thing can be written like so (which uses 19 bytes):

```
a%=3000+a%*100:&,66
```

Based on the value of `a%` (including `0`), `&,66` will `goto` lines starting at 3300 in increments of 100 (`0`=3000, `1`=3100, `2`=3200, `3`=3300, etc.)

4.67. `&,66,1` Calculated `gosub`

`gosub` line number held in `a%`. If you have 13 4-digit `gosub` targets on a line, this can save a fair amount of RAM (and BASIC interpretation time).

Example:

Instead of writing this (which uses 73 bytes):

```
on a% gosub 3000,3100,3200,3300,3400,3500,3600,3700,3800,3900,4000,4100,4200,4300
```

The same thing can be written like so (which uses 38 bytes):

```
a%=abs(val(an$)):if a%<15 then a%=3100+a%*50:&,66,1
```

Based on the value of `a%` (including `0`), `&,66,1` will `gosub` lines starting at `3100`, in increments of 50 (`0`=3100, `1`=3150, `2`=3200, `3`=3250, etc).

```
a%=3:a%=3100+a%*50:&,66,1
```

would `gosub 3250`.

4.68. `&,67` copyrite

4.69. `&,68` struct

Certain sub-functions of `&,60` are re-directed here.

4.70. &,69 Display String on Console

This function will not draw PETSCII graphics characters properly.

Parameters:

`&,69,column,row,text,color`

`column`: `0-39`

`row`: `0-24`

Upper left of the screen starts at column 0, row 0.

`text`: can be a string (`a$`), literal text (`"Hi there!"`), or a combination of both (`"Hi there, "+a$+"!"`)

`color`: is `1-15` for un-reversed colors. (`0`, black, is excluded—this is the same numbering as MCI colors.)

For reversed colors, add `$80` (or `128` decimal). `$8x` is reverse color *x* (`$81` or `129` is reverse white, `$8f` or `142` is reverse light gray).

Example:

im

```
13 &,69,4,21,left$(" "+cm$+"{21 spaces}",22),$8c
```

1. `&,69,4,21`: position string at column 5, row 22
2. `left$(" "+cm$+"{21 spaces}",22)`: format the string so it's left-justified in the 22-character `Area` window
3. `,$8c`: draws the string in reverse (`$8`) in color `$c` (decimal 13, light green)

The module `sub.display` is a good example of using `&,69`.

4.71. &,70 Position Terminal Cursor

Parameters:

`&,70,column,row`

Upper left corner of the screen is `0,0`.

`column`: 0-39

`row`: 0-24

i.IM

```
3350 ... &,70,.,n/2+8.5:&"{white}"
```

Chapter 5. Structures

> ℹ 8/1/2022: Documentation under heavy development and discovery. Feedback welcome.

Structures (or *structs* for short) allow you to access and manipulate the memory used by arrays at machine language speeds.

Programmed properly, structs save RAM compared to having multiple string and numeric arrays defined to, say, hold information about a U/D library:

- filename and filetype
- the block size
- times downloaded
- upload date and last download date

You can now hold all this information within a *single* struct array, performing:

- searches
- sorts
- filtering the list of files based on a substring match
- and more.

You can store multiple types of data in a single struct—each category is stored in a separate “column” called a *field*--and perform operations on the data using `&,60` or `&,68`, and various sub-commands.

Performing a struct search operation is much faster than searching through an array in BASIC with a `for…next` loop.

5.1. Static Arrays

The arrays are considered *static* because, like static strings embedded in BASIC text (such as `a$="hello"`), the size of individual fields can’t change once the struct is defined.

> ℹ Currently there’s no option to resize a struct without first copying data to another struct (accomplished with `&,60,10` Copy Record).

There are functions to:

- put and get strings, and Image BBS-style 11-digit dates
- sort and filter data

- test and collect information from the struct, then put the results in another array
- copy pieces of information from one part of the struct to another
- load and save structs

FIXME more functions

Elements are the individual “boxes” in the array that data is held in.

Strings can be stored in either a floating point [like `a()` or `b()`], or integer [like `a%()` or `b%()`] array.

- Two bytes of a string’s text can be stored per element of an integer array
- Five bytes of a floating point array, or 4 bytes… FIXME

Table 10. Representation of a Sample Two-Dimensional Integer Array

	Record (0,*x*)		
Field (*0*,0)	`x%(0,0)`	`x%(0,1)`	`x%(0,2)`
Field (*1*,0)	`x%(1,0)`	`x%(1,1)`	`x%(1,2)`
Field (*2*,0)	`x%(2,0)`	`x%(2,1)`	`x%(2,2)`

It is suggested that you use numeric instead of string arrays, since this will allow you to both access the elements as numeric data, as well as put and get strings.

Floating point arrays use 5 bytes per element. Integer arrays use 2 bytes per element. When deciding to use structs, you should determine what types of data you will need to store, and how much memory that data will require.

Example 4. User Data, Part 1

As a running example, let’s design a struct to hold a user’s ID#, handle, and password. A module will be written to store, edit, and retrieve data to/from this struct.

- The ID is an integer type (never > 32767), requiring 2 bytes.
- The handle can be up to 20 characters.
- The password can be up to 15 characters.

Since the integers store in 2 bytes, the total number of bytes needed is 37 (2 + 20 + 15). Thirty-seven bytes would require either:

- 19 integer elements (2 bytes per element × 19 elements = 38 bytes)

or

- 8 floating point elements (5 bytes per element × 8 elements = 40 bytes).

Now that you have the basic concept of the struct, let’s look in a little more detail. Here is a byte-by-byte map of the struct we designed.

Table 11. Sample "User Data" Struct Layout

Element Position	Byte Position	Data	Type	Bytes Used
Element 0	Bytes 00-01	ID#	Integer	2 bytes
Element 1	Bytes 02-21	Handle	String	20 bytes
Element 11	Bytes 22-36	Password	String	15 bytes
Element 18½	Byte 37	*unused*	*n/a*	1 byte

Table 12. "User Data" Struct Data Storage

Element	u%(0,0)	u%(0,1)	u%(0,2)	u%(0,3)	u%(0,4)	u%(0,5)	u%(0,6)	u%(0,7)	u%(0,8)	u%(0,9)	u%(0,10)
Byte Pos	00 01	02 03	04 05	06 07	08 09	10 11	12 13	14 15	16 17	18 19	20 21
Data	ID#	Handle *(20 bytes)*									
Bytes	0 1	P I	N A	C O	L A	D A	*x x*	*x x*	*x x*	*x x*	*x x*

Element	u%(0,11)	u%(0,12)	u%(0,13)	u%(0,14)	u%(0,15)	u%(0,16)	u%(0,17)	u%(0,18)	
Byte Pos	22 23	24 25	26 27	28 29	30 31	32 33	34 35	36	37
Data	Password *(15 bytes)*								*unused*
Bytes	P A	S S	W O	R D	*x x*	*x x*	*x x*	*x*	*unused*

Because the unused byte 37 is *not* on an even element boundary (the previous element is an odd number of bytes), it cannot be used.

Since the ID# is an integer anyway, it would be best to use an integer array. The definition would look like this:

```
dim u%(18)
```

Remember that arrays start at element 0! There are 19 bytes in this struct, 0-18.

Of course, you may want to store more than one of these records in memory. To do so, you would need a 2-dimensional array. (Suppose that *x* is the number of records you want.) This would change the `dim` statement to:

```
dim u%(18,x-1)
```

5.2. Some New Terminology

To refer to data in a struct, and hopefully reduce confusion about “elements” and “bytes,” the following terminology will be used:

- The first number in the array notation is the *field number* (like a field within a record of a RELative file). It’s reccommended to be an even number since integers occupy at least two bytes.
- The second number is the *record number*. When the size of the struct is `dim`ensioned, you use this value to address individual records within the struct.

> *Record* and *field* are specified in what most people and programs would consider reverse order (in a database, a record is composed of fields of information). Sorry, there’s no way around this (that we’re aware of).

TODO: a visualization of fields in a record.

Table 13. Fields in Records

`u%(field,record)`	**Fields 0-1**		**Field 2**	**Field 3**	**Field 4**
Record 0 [`u%(0,0)`]	*— configuration information —*				
Record 1	a	b	c	d	e
Record 2	f | g		h	i	j
Record 3	k | l		m	n	o

> Record `0`, field `0` [*e.g.*, `u%(0,0)`] is often used to hold the number of records in the struct. Record `0` may hold additional information in other fields during the lifetime of the struct.

5.3. Using Structs

Now down to the important part: how to use all of this! The struct system is called with either `&,60,sub-function,…` or `&,68,sub-function,…`.

There are currently 14 sub-functions supported by the struct routines. They are documented below.

5.4. Numeric Values and Structs

The array used with structs is either an integer or floating point type. To put numeric values into—or get numeric values from—a struct requires no special struct calls.

You may use code similar to the following examples:

Table 14. Get Number From and Put Number Into Struct

Get value	Put value
`f=a%(3,3)`	`a%(3,3)=20`

Integer arrays can store values from `-32768` to `32767`.

5.5. `&,60,0` Put String

Copies a specified string variable (up to a specified length) into a field of a record of a struct.

Syntax

`&,60,0,` *length*, *struct%(field, record)*, *string$*

Parameters

length: the maximum string length to put into the record.

struct%(field, record): the struct name, field and record you're putting the string into.

string$: the string variable name to assign the struct data to.

Example 5. Put String

```
&,60,0,20,u%(1,1),na$
```

1. Put a string:

&,60,0,20,u%(1,1),na$

2. of up to 20 bytes:

&,60,0,**20**,u%(1,1),na$

3. from the `u%()` array (field 1, record 1):

&,60,0,20,**u%(1,1)**,na$

4. into the string variable `na$`:

&,60,0,20,u%(1,1),**na$**

TODO: test if putting string longer than *length* into struct is truncated — it should be.

Returns

`?type mismatch  error`: if the parameter *string$* is not a string variable FIXME

5.6. `&,60,1` Get String

This copies data from a struct into a specified string variable.

Syntax

`&,60,1,`*length, struct%(field, record), string$*

Parameters

The parameters *length*, *struct%(field, record)*, and *string$* are the same as `Put String` above.

Example 6. Get String

```
&,60,1,20,u%(11,2),a$
```

> ❗ Feedback wanted: which is better, format 1 or format 2? But that’s just, like, your opinion, man.

Example 7. User Data, Part 2

In our earlier example user data struct, to access the third user’s password, you would do this:

```
&,60,1,20,u%(11,2),a$
```

Table 15. Format 1

Parameter	Purpose
`&,60,1,`…	Get a string…
`20,`…	of at most 20 bytes…
`u%(11,2),`…	from the array `u%()`, record `2`, field `11`…
`a$`	into the string variable `a$`.

Format 2

`&,60,1,``20,u%(11,2),a$`

1. Get a string…

`&,60,1,`**`20,`**`u%(11,2),a$`

2. of at most 20 bytes…

`&,60,1,20,u%(11,2),a$`

3. from the array `u%()`, record `2`, field `11`…

`&,60,1,20,u%(11,2),a$`

4. into the string variable `a$`

5.7. `&,60,2` Load Struct from Disk

Loads the specified struct on disk into an array.

Syntax

`&,60,2,0,` *struct%(field, record), filename$, device*

Parameters

`&,60,2,0,`: Required parameters.

struct%(field, record),: FIXME

filename$,: FIXME

device: FIXME

Setup

Assign the variable `dr` to the Image drive number desired, and `gosub 3`. This returns *device* (`dv%`).

(For our example, we’ll set `dr=6`, since `u.` files live on Image drive 6.)

```
dr=6:gosub 3
```

This also returns the drive prefix, `dr$`.

Example 8. Load Struct from Disk

```
&,60,2,0,u%(0,0),dr$+"u.handles",dv%
```

Parameters

1. Load a struct:

`&,60,2,0,u%(0,0),dr$+"u.handles",dv%`

The `0` is believed to be a necessary but ignored parameter.

2. Use the u%() array (load to record 0, field 0):

```
&,60,2,0,u%(0,0),dr$+"u.handles",dv%
```

> You do not have to load the file at the start of the array. The starting record and field are specified in the array notation. This example loads the file u.handles into the u%() array, starting at the beginning of the array (0,0). It could load starting at (0,5) — record 5, field 0 — or anywhere else you want, as long as it is within the bounds of the struct's dimensions.

3. Use the drive prefix dr$, plus the fictitious "u.handles" filename:

```
&,60,2,0,u%(0,0),dr$+"u.handles",dv%
```

4. dv% is the device number to load the struct from:

```
&,60,2,0,u%(0,0),dr$+"u.handles",dv%
```

5.8. &,60,3 Save a Struct to Disk

This saves a struct to a specified disk file.

Syntax

&,60,3,0, *struct%(field, record), filename$, device*

Setup

TODO use include:: from &,60,2 setup

Parameters

&,60,3,0, *struct%(field, record), bytes, filename$, device*

The parameters *struct%(field, record)*, *bytes*, *filename$*, and *device* are the same as in previous commands.

The starting record and field numbers to save are specified by the numbers in the array notation.

Example 9. Save Struct to Disk

```
&,60,3,0,u%(0,0),38*3,dr$+"u.handles",dv%
```

Calculating Struct Size to Save

The number of bytes should be calculated using the formula:

bytes per record × *number of records*

> (There are 38 bytes per record × 3 records in the example.)
>
> Don’t forget: records start at `0`!

The starting record and field is specified with (as above) `u%(0,0)`.

1. Save a struct:

`&,60,3,0,``u%(0,0),3*38,dr$+"u.handles",dv%`

2. The starting element is specified with *struct%(field, record)*:

`&,60,3,0,`**`u%(0,0),`**`3*38,dr$+"u.handles",dv%`

3. *bytes*: the number of bytes the struct occupies is the number of records multiplied by the bytes per record. In our example, 3 records × 38 bytes:

`&,60,3,0,u%(0,0),`**`3*38,`**`dr$+"u.handles",dv%`

4. drive prefix `dr$` + filename (the theoretical `u.handles`):

`&,60,3,0,u%(0,0),3*38,`**`dr$+"u.handles",`**`dv%`

5. device `dv%`, set by `gosub 3` before the struct save call

5.9. `&,60,4` Put Date

Put an 11-digit date string into a struct (converted from 6 bytes as stored in Binary Coded Decimal).

Syntax

`&,60,4,0,` *struct%(field, record), string$*

struct%(field, record): struct name, record and field to store date in

Parameters

string$: the 11-digit date string (either a literal string or string variable?) FIXME

Returns

`?illegal quantity error` if the date string is not 11 digits

Example 10. Put Date

```
an$="10412208234":&,60,4,0,u%(3,0),an$
```

TODO: Explain example.

Details: Binary Coded Decimal

Structs store an 11-digit date in 3 elements (6 bytes) using Binary Coded Decimal (BCD) format. Two decimal digits are stored per byte, using the high and low *nybbles* (*i.e.*, 4-bit halves of an 8-bit number).

an$="10412208234":&,60,4,0,u%(0,1),an$

Element	`u%(0,1)`		`u%(0,2)`		`u%(0,3)`		*unused*
Binary	`%0001 %0000`	`%0100 %0001`	`%0010 %0010`	`%0000 %1000`	`%0010 %0011`	`%0100`	`%xxxx`
Decimal	`1 0`	`4 1`	`2 2`	`0 8`	`2 3`	`4`	`x`

5.10. `&,60,5` Get Date

Convert a 6-digit Binary Coded Decimal (BCD) date string (as shown above) to the 11-digit format as shown above.

Parameters

`&,60,5,0,` *struct%(field, record), string$*

Parameter	Purpose
`&,60,5,0`,…	Get date call. `0` seems to be an ignored but necessary parameter.
`struct%(field, record)`,…	struct name, field, record…
string$	…string variable to hold the converted 11-digit date and time

Example 11. Get Date

```
&,60,5,0,u%(0,1),an$:&,15:&an$
```

1. `&,60,5,0,`: Get a date string…
2. `u%(0,1),`: …from the struct `u%()`, field `1`, record `0`…
3. `an$:` …into `an$`.
4. `&,15`: Convert `an$` into a long date string.
5. `&an$`: Display the long date string.

Result

TODO: finish the output

5.11. `&,60,6` Scan Struct

Scan through a field in a struct, testing whether various conditions are true on variables. If the condition is true, perform an operation on another field in the struct.

Syntax

`&,60,6,` *num*, *command*, *a%(a,b)*, *b%(a,b)*, *size*, *bits*, *test*

Example: i.GF

```
3166 a%=0:if s%(0,0) then:&,60,6,s%(0,0),0,s%(0,1),s%(1,1),80,1,2^ac%
```

Statement	Variable	Purpose
`if s%(0,0) then…`	*n/a*	There is an implied `if s%(0,0)<>0` here, meaning “if the record count is non-zero, then…”
`&,60,6,`…	*n/a*	scansum
`s%(0,0),`…	*num*	for the record count
`0,`…	*command*	`0`: 2-byte `and` between bits in `s%(0,1)` and `s%(1,1)`? FIXME
`s%(0,1),`…	s%(*field*, *record*)	starting flags element
`s%(1,1),`…	s%(*field*, *record*)	starting object element
`80,`…	*size*	each record is 80 bytes wide
`1,`…	*bits*	set bit 1 on … if *command* returns zero? FIXME
`2^ac%`	*test*	access level

Example 12. i.MM.load

```
4106 &,60,6,x1%(0,0),0,x1%(0,1),x1%(1,1),36,4096,2^ac%
4108 &,60,6,x1%(0,0),5,x1%(0,1),x1%(0,1),36,8192,f
4110 &,60,6,x1%(0,0),7,x1%(0,1),x1%(0,1),36,16384,id
```

`num`: # of fields in the struct to scan

`bits`: the bits to set if *test* is true

`flag%(field, record)`: the struct name, record and field on which to set `bits` if `test` is true.

record may be a dummy parameter, more tests needed.

`scan%(field, record)`: struct name, record and field to scan `size`: record size in bytes

`command`: command number as listed in this table:

Table 16. Scan Struct Commands

Num	Command	Add If Result
0	2 byte `and`	not equal to `0`
1	2 byte `and`	equal to `0`
2	2 byte `cmp`	less than (`<`)
3	2 byte `cmp`	greater than or equal to (`>=`)
4	date `cmp`	date is less than (`<`)
5	date `cmp`	date is greater than or equal to (`>=`)

Num

Command number

Command

How to compare the two objects:

- `and` does a logical and with the bits FIXME
- `cmp` compares values

Add If Result

Add this record (field?) to the FIXME only if *object* meets the command's criteria

test: the object to test for (apparently can be either a variable or a number, maybe the byte number?)

Example 13. `i.UD` *from Image 2.0*

This is still being researched.

The following code scans the U/D directory for entries which have an upload date older than `ld$`, setting bits `$4f` on `ud%(3,1)` (if the entry matches?):

i.UD:

```
3950 &,60,6,rn,$4f,ud%(0,1),ud%(3,1),60,4,ld$:b%=a%
```

`rn`: highest record number to scan in the directory struct

`$4f`: (`%0100 1111` in binary) FIXME still researching the purpose of this

`ud%(0,1)`: FIXME

`ud%(3,1)`: Upload date

`60`: record is 60 bytes wide

`4`: date comparison, `<` (less than)

`ld$`: the comparison object (last call date). Can apparently be a string name, or number of an array?

Returns

`a%`: count of fields the comparison returns as matching `test`.

`b%(a,b)`: the array containing the comparisons matching `test`.

5.12. `&,60,7` Sort Struct

Sort a string array (only two-dimensional?). Does not work with numeric arrays.

Syntax

`&,60,7,0,` *a$(a, b), start*

Parameters

a$(a,b): String array to sort

start: Element to start sorting at?

Example 14. i/lo/tt maint

```
4016 for i=1 to 8:&".":&,60,7,0,a$(p+1,i),n-p:next:p=n-10
```

5.13. `&,60,8` Scan Numbers

Scan through a specified field in a struct for non-zero values. `a%` returns how many non-zero values there are. The list of non-zero values are returned in the specified array.

Syntax

`&,60,8,` *number, size, access, struct%(field, record), result%(1), start*

Parameters

`number`: number of records to scan

`size`: size of the record, in bytes

`access`: access level to filter results by (in bits?)

`struct%(field, record)`: the struct, record and field to scan `result%(1)`: the single-dimension array to put the results in. `1` seems to be a dummy parameter: ignored, but necessary to be interpreted as a valid array reference.

`start`: record to start scanning at

Example 15. i/MM.load

```
4112 &,60,8,x1%(0,0),36,8192+16384,x1%(0,1),x2%(1),1:x2%(0)=a%
```

1. `&,60,8`: Scan Numbers sub-command
2. `rn`: Scan through `rn` records
3. `60`: the struct is 60 bytes per record
4. `a`: filter by access level `a`
5. `ud%(0,1)`: look in the `ud%(field=0, record=1)` (field *0*=“don’t care?”)
6. `f%(x)`: put non-zero results in the `f%()` array
7. `1`: Start at record 1.

ℹ More research needed. `8192+16384` exceeds the expected access levels of 0-9 (values 1-1023).

Example 2: i.UD

```
3310 &,60,8,rn,60,a,ud%(0,1),f%(1),1:f%(0)=a%
```

FIXME: order of params changed — this is Jack’s struct UD

1. `&,60,8`: Scan Numbers sub-command
2. `rn`: Scan through `rn` records
3. `60`: the struct is 60 bytes per record
4. `a`: filter by access level `a`
5. `ud%(0,1)`: look in the `ud%(field=0, record=1)` (field *0*="don’t care?")
6. `f%(x)`: put non-zero results in the `f%()` array
7. `1`: Start at record 1.

Returns

`a%`: number of results returned, `0`=none.

`a%(a)`: one-dimensional array of results, from `a%(1–a)`

5.14. `&,60,9` Scan Sum

Syntax

`&,60,9,` *number, size, struct%(field, record)*

`number`: number of records to scan

`size`: size of record, in bytes

`struct%(field, record)`: (field="don't care"? FIXME), record to scan

This function call documentation is incomplete.

Example

None yet.

Returns

`a%`: FIXME: total of values in struct?

5.15. `&,60,10` Copy Record

Copy one record from one struct to a record in another struct.

Syntax

`&,60,10,` *size, a1(a, b), a2(a, b)*

Parameters

`size`: size of record

`a1%(a,b)`: source struct `a1%()`, record `b` and field `a`

`a2%(a,b)`: destination struct `a2%()`, record `b` and field `a`

Example 16. i/IM.logon

```
4694 if x<>fb%(.,.) then for a=x to fb%(.,.)-1:&,60,10,60,fb%(.,a+1),fb%(.,a):next
①
```

① `if x<>fb%(0,0)`: if `x` does not equal the number of records in the struct [`fb%(0,0)`], then copy record `a+1` to record `a` in a loop.

5.16. `&,60,11` Scan for String

Scan struct for a string present in a specified field and record. Put results in another specified struct, field and record?

Syntax

`&,60,11,` *num*, *size*, *op*, *str*, *a1%(a,b)*, *a2%(b)*, *start*

Parameters

`num`: number of records to scan

`size`: size of record

`op`: operation:

- `0` specifies a regular compare (a string literal)
- `1` specifies a pattern to match. Here you can use two wildcard characters (like Commodore DOS):
  - `f2` (in quote mode: `I`) is equal to `?`, which specifies any character in its place
  - `f7` (in quote mode: `H`) is equal to `*`, which specifies any characters from this point to the end of the string

`str`: FIXME: string variable/string literal to scan for?

`a1%(a,b)`: source struct *a1%()*, record *b*, field *a*, to scan

`a2%(b)`: target 1-dimension array *a2%()*, dummy element *b*, to put results into

`start`: record to start scanning from

Returns

No info yet.

Example

None yet.

5.17. `&,60,12` Game Scan

Unknown purpose.

Syntax

`&,60,12,` *count*, *size*, *a$*, *a%(a,b)*, *b$*

Parameters

`count`: how many records to scan?

`size`: size of the record to scan

`a$`: a string to search for?

`a%(a,b)`: `a%()`: struct name, `a`: field and `b`: record to scan

`b$`: ?

Example

None yet.

5.18. `&,60,13` Text Read

Not sure yet. Read a file into a struct?

Syntax

`&,60,13,` *number, reclen, scan(), bits, text(), strlen*

Parameters

number,: count of lines to read?

reclen,: record length?

scan(),: ?

bits,: ?

text(),: ?

strlen: ?

Example

None yet.

Chapter 6. Variables

Within Image BBS, there are certain variables which can be considered “reserved.” This does not mean that you cannot use them, *per se*, but that they can only be used for specific purposes:

- Some variables may be used any time, but have a specific purpose.
- Some variables can be used with certain subsystems, but not with others.
- Some variables may be used anywhere, but change continually.

This is explained in detail in the following paragraphs.

An example of a variable used for a specific purpose is `na$`. This variable is used to print the handle of the user online. Storing something for a module in this variable would cause an undesirable effect (modifying the user’s handle). Basically, these types of variables are used to control system statistics and are best left alone, only to be used to output information.

Some variables are used as interfaces between the BASIC and ML, an example being `pl`. Setting `pl` to `0` will cause all user input to be in the form of upper- and lowercase characters. When you set `pl` in BASIC, it causes the ML input routines to accept both uppercase and lowercase characters.

The main variables that can be used sometimes are arrays. Depending on which subsystem you are in, the arrays may or may not be in use. If they are not in use, it is safe to use them. The only exceptions to this is `st()`, `dv%()` and `tt$()`.

- `st()` holds the Board Activity Register stats; changing the values of this array should be reserved for updating the BAR stats.
- `dv%()` is the system device designator. Altering this will change the device accessed by the system.
- `tt$()` cannot be used in a module that calls the editor, unless you want to edit existing text in that array. All text stored in the editor is put into `tt$()`.

Some variables are intended to continually change. These include variables that will print text to the screen and modem, as well as variables that form links between the BASIC and ML portions of the program. Examples are `a$` and `an$`:

- When used in conjunction with `&`, the value of `a$` will be printed to the screen and modem.
- All responses entered at a prompt will be stored in `an$`.

These variables have a set purpose, but are intended to change.

6.1. Variable

This is the variable’s name.

6.2. Type

`BBS`: this is a BBS statistic or BBS-maintained variable, it could be saved to `e.data`.

`User`: this is a user statistic, saved to `u.config`.

6.3. Use?

✓ indicates you can assign a value to it within your own modules.

⊗ indicates the variable is maintained by either the BBS ML or BASIC modules. User data corruption or other unwanted side-effects may occur if this variable is reassigned.

? means the variable may be used in some circumstances, but not in others.

6.4. Purpose

An explanation of what the variable does.

6.5. Integer

Variable	Type	Use?	Purpose
`a%`			
`ac%`	BBS	✓	User access level, `0-9`. **Read:** Get user’s access level. **Write:** Set user’s access level.
`af%`			Access flag?
`ao%`	BBS	✓	Old access level (“Access Old”). If `ac%<>ao%`, the user’s access level has changed during the call.
`cd%`	User	✓	Last call carrier drop flag.
`co%`	User	✓	Computer type: `co$(co%)` is the computer name.
`ct%`	User	✓	Calls today.
`d1%`	BBS	✓	Currently active device number.
`d2%`	BBS	⊗	Currently active drive/LU number. FIXME: duplicate?
`d3%`	BBS	⊗	Currently active drive/LU number. FIXME: duplicate?
`da%`	User	✓	Downloads allowed per day.
`db%`	User	✓	Downloaded blocks.
`df%`	BBS	✓	Default color for text.
`kp%`	BBS	⊗	ASCII value of keypress?

Variable	Type	Use?	Purpose
`ll%`	User	✓	Line length (width of the user’s screen in columns)
`mn%`	BBS	⊗	Minute of the day: `1-1440`
`p1%`	BBS	⊗	Prime Time: Minutes allowed during prime time.
`p2%`	BBS	⊗	Prime Time: Starting hour.
`p3%`	BBS	⊗	Prime Time: Ending hour.
`pt%`	BBS	⊗	Prime Time: Active flag
`tc%`	User	⊗	Total calls to the system by the user online

6.6. Floating Point

Variable	Type	Write	Purpose
`am`	BBS	⊗	AutoMaint flag.
`bd`	BBS	✓	BBS boot drive, used only during initialization.
`bf`	BBS		Temporary blocks free count.
`ca`	BBS		Total calls to the BBS.
`cr`	User	✓	Amount of credits the user has.
`el`	BBS	✓	Line number an error has occurred on.
`em`	User	✓	Expert mode flag: `0`=disabled, `1`=enabled
`is`	BBS	⊗	Image `sub.*` module stack depth counter.
`l1`	BBS	⊗	BBS reservation:
`l2`	BBS	⊗	BBS reservation:
`l3`	BBS	⊗	BBS reservation:
`lp`	BBS	✓	Read: `&,5` (get ML version data) `lp` returns the ML major/minor version number, *e.g.*, `1.3`. Use: Disable or enable word-wrap for `&` text output. `lp=0`: disable word-wrap, `lp=1`: enable word-wrap
`nt`	BBS	⊗	Network transfer flag: `0`=no transfer occurring, `1`=in NetMaint (NMauto) mode.
`pf`	BBS		General Files directory stack depth counter.
`pm`	User	⊗	Prompt Mode flag: `0`=disabled, `1`=enabled
`uc`			
`uh`	BBS		
`ul`	BBS		

Variable	Type	Write	Purpose
`ur`	BBS		
`zz`	BBS	⊗	Pseudo-local mode flag: `0`=disabled, `1`=enabled

6.7. Strings

`b$`-`z$` are work variables used throughout the BBS by different subsections. They are available for use and may be read and written freely.

Some specific information about certain variables is outlined below.

Variable	Type	Use?	Purpose
`a$`	BBS	✓	Output text using `a$="text":&`. General-purpose work variable.
`ag$`	BBS	⊗	Access group information, including 4 control characters and access group name. (Also MCI variable `£vm`.)
`ak$`	BBS	✓	A horizontal line 2 characters less than the user’s screen width. (Also MCI variable `£vj`)
am$			
`an$`	BBS	✓	Character input from `£gx`, strings input from `£ix` or `&,1`. `&,15,x` (convert `an$`): perform various conversions on `an$`. (Also MCI variable `£v7`.)
`bd$`	BBS	✓	Boot drive partition/LU number. Used once during `im` initialization.
`bn$`	BBS	⊗	BBS name. (Also MCI variable `£v5`.)
bs$ - used once, line 3100			
`c1$`	BBS	✓	Chat mode entry message.
`c2$`	BBS	✓	Chat mode exit message.
`c3$`	BBS	⊗	`Returning To The Editor` message (hard-coded, `im` line .
`cc$`	BBS	⊗	2-character system identifier, sometimes shown with user ID. (Also MCI variable `£vn`.)

Variable	Type	Use ?	Purpose
`ch$`			Copy of co$?
`cm$`	BBS	✓	Current Message, displayed in the `Area` sysop console screen mask. (Sometimes used for debugging information in `e.errlog`.)
`co$`	BBS	✓	User's computer type, displayed in 16-character programmable window using `&,9,36`. Equivalent to `co$(co%)`.
`d1$`	BBS	⊗	Current time and date information in 11-digit format. (Also MCI variable `£v0`.)
`d2$`	BBS	⊗	Time and date of last logoff, or Library name at entry. (Also MCI variable `£v8`.)
`d3$`	BBS	⊗	Handle of last user on the system. (Also MCI variable `£v9`.)
`d4$`	BBS	⊗	Name of current ML protocol in memory. (Also MCI variable `£vl`.)
`d5$`	BBS	⊗	True last call date of user online in 11 digit format. (Also MCI variable `£vk`.)
`d6$`	BBS	⊗	Logoff time of last user.
`dd$`	BBS	⊗	System identifier + user ID number
`dr$`	BBS	⊗	Currently active drive/LU number + `:`
`ef$`	BBS	⊗	ECS command flags.
`ep$`	BBS	⊗	ECS command password.
`f1$`--`f8$`	BBS		Programmable function key definitions. Strings must end in null byte (`nl$`).
`ff$`	User	⊗	Real first name of user online.
`fl$`	User	⊗	20-character string which determines the user's status flags.
`hx$`	BBS	⊗	16 hexadecimal digits: `"0123456789abcdef"`.
im$			
in$			
`i1$`	BBS	⊗	Access level + handle of the sysop.
`jn$`	User	✓	`dim`*ensioned but unused?* Sub-board "joined read" string from pre-TurboREL 1.2 SB subsystem.
l1$			

Variable	Type	Use?	Purpose
l2$			
l3$			
`ld$`	User	⊗	Last call date of user online in 11-digit format. Used to determine whether a message is new or not.
`ll$`	User	⊗	Real last name of user online.
lm$			
`lt$`	BBS	⊗	Logon time of user online in 11-digit format.
`ml$`	BBS	⊗	Filename of current ML module in memory.
`mp$`	BBS	⊗	More prompt text: `···More (Y/n)?` (hard-coded in `im`, line FIXME)
`mt$`	BBS	⊗	modem setup?
`na$`	BBS	⊗	Handle of current caller. (Also MCI variable `£v2`)
`nl$`	BBS	⊗	Null character [`chr$(0)`]
`nm$`	BBS	⊗	Last network sort time/date in 11-digit format.
`p$`	BBS, ML	✓	Current prompt text.
p1$			
p2$			
`ph$`	User	⊗	E-mail address of current user online. (Also MCI variable `£v4`)
`po$`	BBS		Text for system main level prompt.
`pp$`	BBS	⊗	System password (change with `PC` command) FIXME: still used?
`pr$`	BBS		Name of current `pr`ogram (module) in memory.
pu$			
`pw$`	User	⊗	Password of current online user
`qt$`	BBS	⊗	Quotation mark [`chr$(34)`].
`r$`	BBS	⊗	Return character [`chr$(13)`]
`rn$`	User	⊗	Real name of user online (`ff$+" "+ll$`) (Also MCI variable `£v3`)
sb$			
`sy$`	BBS	⊗	Current subsystem active.
`ti$`	BBS	✓	C= Time-of-day clock

Variable	Type	Use?	Purpose
tk$			
tt$			
tz$			Time zone
u$	BBS	⊗	Reserved for command stacking.
uf$			User flags.
uu$			Command stacking.
w$			Word-wrap input.
x$	BBS	⊗	System drive/LU designators FIXME?
z1$			only during config
z2$			only during config
z3$			only during config

6.8. String Arrays

Variable	Type	Use?	Purpose
co$(9)	BBS	✓	Text of computer types.
hs$(10)	BBS	⊗	User command history stack.
is$(10)	BBS	⊗	sub.* module call stack.
pf$(10)	BBS		General File directory names stack. GF section remembers which menu level you were at after quitting a module.
tt$(254)	BBS	?	Text entered into text editor. This array can be used in modules not using the text editor.

6.9. Floating Point Arrays

Variable	Type	Use?	Purpose
bf(6)	BBS	⊗	Blocks free on system disks.

6.10. Image 1.2 Arrays

Image 1.2 Arrays

bb$(31)

dt$(31)

ed$(61)

nn$(61)

a%(61)

c%(61)

d%(61)

e%(31)

f%(61)

ac%(31)

so%(31)

Chapter 7. POKEs and Memory Locations

`poke`s control various flags maintained by the Image BBS machine language (ML).

Decimal	Hex	ML Label	BASIC Variable	Purpose
`830`	`$33e`			Time limit
`951`	`$3b7`	`modclmn`	`ll%`	Terminal width in columns.
`970`	`$3ca`	`usrlin`	*n/a*	Number of lines output. If the **More Prompt** is enabled, compare `usrlin` to `usrlinm` to know when to display the prompt. Line output count can be reset with `poke 970,0`.
`971`	`$3cb`	`usrlinm`	`mp%`	Terminal height in rows.
`1010`	`$3f2`	`timeset`	*n/a*	Time set flag. `0`=Clock has been set, stop flashing bottom status line.
`17138`	`$42f2`	*n/a*		Password mask character.
`53252`	`$d004`	`llen`	*n/a*	Control input line length.

💡 `poke`s could go away in the future, in favor of an interface table. This replaces using a memory location with a number in the interface table.

Instead of using `poke 53252,22`, the call would be similar to `&,21,0,22`.

Refer to the `&,20`: Read from Interface Table or `&,21`: Write to Interface Table commands for more information.

Chapter 8. Machine Language Entry Points

Here is a listing of ML modules and their entry points.

This section is currently undergoing research.

8.1. Protocols

`&,16,4,x`: `getflag`

`sub.protos`: This returns the value of `defflag` from the protocol. Its purpose is currently unknown.

`&,16,5,x`: `getflag`

This sets the value of `defflag` from the protocol. Its purpose is currently unknown.

8.1.1. `++ index`

This module handles the `u.config` (user log) and `u.weedinfo` (user weed info) files.

Function	Label	Parameter(s)	Returns
`&,16,0`	`find`	`an$`=user name	?
`&,16,1`	`loadindex`	`a$`=filename	?
`&,16,2`	`saveindex`	`a$`=filename	?
`&,16,3`	`makeindex`	`a$`=filename, `b%`=?	?
`&,16,4`	`instindex`	`a%`=id	?
`&,16,5`	`deltindex`	`a%`=id	?
`&,16,6`	`nextindex`	?	?
`&,16,7`	`setcrskip`	?	?
`&,16,8`	`findindex`	`a%`=id	`a%`=0: not found

8.1.2. `++ punter`

Function	Purpose	Returns
`&,16,0`	multi-download	
`&,16,1`	multi-upload	
`&,16,2`	multi-download + header	
`&,16,3`	multi-upload + header	

Function	Purpose	Returns
`&,16,4`	setflag	`a%`
`&,16,5`	getflag	`a%` (are these flags whether the protocol supports multi-file transfers?)

8.1.3. `++ reader`

This module can display PRG files.

Function	Purpose	Returns
`&,16,0,ll%`	Detokenize BASIC file, use column width `ll%`	

8.1.4. Graphic Menu

The ML module `++ menu2` handles adding menu items, hotkeys, displaying the menu and passing the menu item selected back to BASIC.

`&,16,0`: Add menu string

Parameters:

`&,16,0,?,menu%(element,byte),"hotkey(s)","prefix_text?","menu_item_text"`

`?`: `42`: draw two-column menu

`menu%(element,byte)`: struct to put menu text in

`hotkey(s)`: One (`B`) or two (`/B`) keys to type to select this menu item

`prefix_text`:

`menu_item_text`: Text the user sees for this menu item.

Example:

i.GF:

```
3518 if pf>1 then:&,16,0,42,q%(0,n),"/B","Op ","Return to Previous Menu":n=n+1
```

i.IM

```
&,16,.,34,m%(.,1),"A","","Macros Editor"
```

Returns:

`a%`: which item was selected

`&,16,1`: Add string

Parameters:

Unknown.

`&,16,2`: Use menu

Parameters:

`&,16,2,?,menu%(element,byte),item_count?,?,?,menu_height?,?,?,?`

FIXME

Examples:

```
xxxx &,16,2,42,q%(0,0),n,36,1,n,6,6,0:a$=chr$(q%(3,a%) and 255)
```

```
xxxx &,16,2,34,m%(.,1),n,17,2,n/2+.5,2,6,.
```

Returns:

`a%`: item number selected

`&,16,3`: ?

Chapter 9. Lightbar Reference

9.1. Lightbar Numbering

Table 17. Lightbar, page 1

Title	Sys	Acs	Loc	Tsr	Cht	New	Prt	U/D
Decimal	00 01	02 03	04 05	06 07	08 09	10 11	12 13	14 15
Hex	$00 01	02 03	04 05	06 07	08 09	0a 0b	0c 0d	0e 0f

Table 18. Lightbar, page 2

Title	Asc	Ans	Exp	Unv	Trc	Bel	Net	Mac
Decimal	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31
Hex	$10 11	12 13	14 15	16 17	18 19	1a 1b	1c 1d	1e 1f

Table 19. Lightbar, page 3

Title	Chk	Mor	Frd	Sub	Res	Mnt	Mnu	Xpr
Decimal	32 33	34 35	36 37	38 39	40 41	42 43	44 45	46 47
Hex	$20 21	22 23	24 25	26 27	28 29	2a 2b	2c 2d	2e 2f

Table 20. Lightbar, page 4

Title	Em3	Sc2	Scp	Alt	Trb	DCD	DSR	$3e
Decimal	48 49	50 51	52 53	54 55	56 57	58 59	60 61	62 63
Hex	$30 31	32 33	34 35	36 37	38 39	3a 3b	3c 3d	3e 3f

Table 21. Lightbar, page 5

Title	$40	$42	$44	$46	$48	$4a	$4c	$4e
Decimal	64 65	66 67	68 69	70 71	72 73	74 75	76 77	78 79
Hex	$40 41	42 43	44 45	46 47	48 49	4a 4b	4c 4d	4e 4f

Table 22. Lightbar, page 6

Title	$50	$52	$54	$56	$58	$5a	$5c	$5e
Decimal	80 81	82 83	84 85	86 87	88 89	90 91	92 93	94 95
Hex	$50 51	52 53	54 55	56 57	58 59	5a 5b	5c 5d	5e 5f

Table 23. Lightbar, page 7

Title	$60	$62	$64	$66	$68	$6a	$6c	$6e
Decimal	96 97	98 99	100 101	102 103	104 105	106 107	108 109	110 111
Hex	$60 61	62 63	64 65	66 67	68 69	6a 6b	6c 6d	6e 6f

Table 24. Lightbar, page 8

Title	`At1`	`At2`	`At3`	`At4`	`At5`	`At6`	`At7`	`At8`
Decimal	112 113	114 115	116 117	118 119	120 121	122 123	124 125	126 127
Hex	$70 71	72 73	74 75	76 77	78 79	7a 7b	7c 7d	7e 7f

9.2. Lightbar Interface: `&,52`

9.2.1. BASIC `&,52` Commands

`&,52,position,mode`

`position` ranges from 0-127 decimal ($00-$7f hexadecimal--`&,52,$30,0`, for example, is allowed).

`mode` is 0-4 as used by BASIC.

`0`	clear checkmark at *position*
`1`	set checkmark at *position*
`2`	toggle checkmark at *position*
`3`	read checkmark at *position*, return status in `a%`: 0=off, 1=on
`4`	move “lit” portion of lightbar to position 0-55 [FIXME: or 1-56?]

9.2.2. Assembly Example

FIXME

Mode 5 is reserved for use by ML routines, and is the equivalent of `&,52,x,3` in BASIC. `ldx` with the flag to check, `jsr chkflags`, and the result (0 or 1) is returned in `.a`.

checkflag.asm

```
ldx #$04    ; lightbar flag number
jsr chkflag ; returns flag status (0=off, 1=on) in .a
bne flag_on
beq flag_off
```

Chapter 10. Memory Map

Author: Ray Kelm

During boot (contents of the ML file):

Memory range	Purpose
`$6C00` - `$6FFF`	wedge code
`$7000` - `$7FFF`	editor code
`$8000` - `$83FF`	garbage collector
`$8400` - `$8DFF`	ECS code
`$8E00` - `$93FF`	struct code
`$9400` - `$97FF`	`swap1` code
`$9800` - `$9BFF`	`swap2` code
`$9C00` - `$9FFF`	`swap3` code

Everything after this point is the same as the next section.

While running:

Memory range	Purpose
`$0800-$0AFF`	RS232 driver
`$0B00-$0B7F`	RS232 input buffer
`$0B80-$0BFF`	RS232 output buffer
`$0C00-$0FFF`	BASIC wedge
`$1000-$12FF`	temporary screen data
`$1300-$130F`	`chktbl`
`$1310-$13CF`	`bartbl`
`$13D0-$13DF`	array pointers
`$13E0-$13EC`	days per month
`$13ED-$13EF`	unused
`$13F0-$144F`	sounds
`$1450-$147F`	net alarms
`$1480-$14FF`	ASCII to CBM translation table
`$1500-$15FF`	CBM to ASCII translation table
`$1600-$161F`	`tblcta1`
`$1620-$163F`	`tblcta2`
`$1640-$165F`	`tblcta3`

Memory range	Purpose
`$1660-$16E0`	alarm table
`$16E0-$16FF`	date buffer
`$1700-$1718`	`lobytes`
`$1719-$1731`	`hibytes`
`$1732-$174A`	`lobytec`
`$174B-$1763`	`hibytec`
`$1764-$177F`	unused
`$1780-$17FF`	`pmodetbl`
`$1800-$27FF`	editor execution location (swapped in when needed)
`$2800-$9FFF`	BASIC program area
`$A000-$BFFF`	BASIC ROM / Image ML routines in RAM
`$C000-$CAFF`	“Protocol” block for loadable ML code
`$CB00-$CCFF`	swapper area
`$CD00-$CDFF`	interface page
`$CE00-$CEFF`	buffer page
`$CF00-$CFFF`	???
`$D000-$DFFF`	I/O memory
`$D000-$DFFF`	Character ROM
`$D000-$DFFF`	editor swap location (code is here when waiting to run)
`$E000-$FFFF`	KERNAL ROM / Image ML “swap” code in ROM
`$E000-$E3FF`	garbage collector swap module
`$E400-$EDFF`	ECS swap module
`$EE00-$F3FF`	Struct swap module
`$F400-$F7FF`	Swap1 swap module
`$F800-$FBFF`	Swap2 swap module
`$FC00-$FFFF`	Swap3 swap module

Chapter 11. File Formats

11.1. Introduction

Option Lists

To show when options are mutually exclusive (there can be only one option chosen from a group), the following notation is used:

[*option 1* | *option 2* | *option 3*]

means that of the three options presented, option 1 *or* option 2 *or* option 3 is saved in that position in the file.

11.1.1. `e.data`

`e.data` is a RELative file containing BBS configuration information, as well as BBS statistics.

The record size is 31 bytes.

Table 25. e.data File Format

Record	Variable	Purpose	Possible Values
1	`ca`	Total calls to the system	—
2-3	—	*unused*	—
4	—	one-time caller weed cutoff, in months	
5-11	—	*unused*	—
12	`ur`	Highest user account # +1	
13-16	—	*unused*	
17	`d3$`	Last caller handle	
18	`pp$`	Sub-board password for non-RELedit systems	
19	—	Date/time of last user logoff	
20	`p1%,p2%,p3%`	Prime Time info	Time allowed per call, Prime Time start, Prime Time end

Record	Variable	Purpose	Possible Values

21	`l2,l2$`	System Reservation Password	[`0`=None | `1`=One Call | `2`=All Calls] , [`^`=No Password | 1-14 Character Password]
			`0,^`: No reservation, no password
			`1,ONCE`: Reserved for one call with password `ONCE`
			`2,ALL`: Reserved for all calls with password `ALL`

22-30	—	*unused*	
31	—	Next available user account #	
32	—	RS232 Interface Type	[`0`=User Port | `1`=SwiftLink]
33-34	—	*unused*	
35	`am$`, `d6$`	Date/time of last log reset	
36	—	*unused*	
37	`y%`	Clock set device	[`1`=Manual | `3`=Lt.Kernal Port 1 | `4`=Lt.Kernal Port 2 | `8-29`=CMD Device #]
38	`lk%`	Lt.Kernal device number	
39	—	Autoweed cutoff, in months	
40	`df%`	Default text color	
41	`a`	Printer secondary address	
42	`a`	Printer line feeds	[`0`=no | `10`=yes]
43	—	Password mask character(s)	
44	—	Log start date	
45	`tz$`	BBS time zone abbreviation/hour offset	*e.g.*, `EST0700`
46	—	*unused*	
47	`bn$`	BBS name	
48	`c1$`	Entering chat message	
49	`c2$`	Exiting chat message	
50	—	Netsub ID	increments by 1 when net post/response made
51	`cc$`	BBS identifier	*e.g.*, `WN`

Record	Variable	Purpose	Possible Values
52	—	System device, drive	For records 52-57, devices and drives are in separate fields of each record (stored as two lines separated by a carriage return).
53	—	E-Mail device, drive	
54	—	Etcetera device, drive	
55	—	Directory device, drive	
56	—	Module device, drive	
57	—	User device, drive	
58	`nc`	Credits for new user	