

AssetPack

Command Set

for SkyBitz/Ametek Integration

**2 December 2019**

**AssetLink Global LLC**

[**www.assetlinkglobal.com**](http://www.assetlinkglobal.com/)

[**service@assetlinkglobal.com**](mailto:service@assetlinkglobal.com)

**+1 (303) 862.8745**

**Fax: +1 (720) 862.3515**

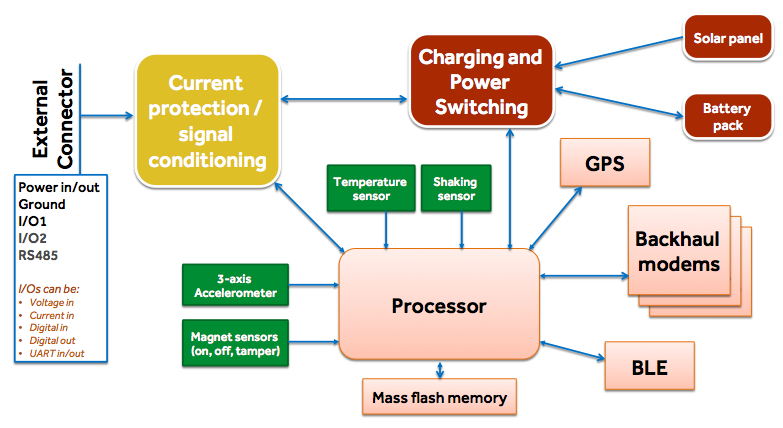
**Revision History**

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date of Release** | **Purpose** |
| Initial Draft | 12/2/2019 | Condensation of internal AP command set for ST9x integration |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Introduction**

This document is extracted from AssetLink internal documents “AssetPack-N Defined-Capability Unit Architecture & Configuration” and “AssetPack Command Set”, to focus on the commands that are useful for the integration of the ST9x product with the AssetPack. The ST9x will use the AssetPack as a commandable message delivery system and a power source.

The full set of AssetPack hardware capabilities may be seen here (not all units have all hardware installed):



All AssetPack hardware versions use a common codebase, and a common set of commands to define and change unit behavior. These commands are central to the operation of the AssetPack:

* All over-the-air commands, use this command set.
* All internal responses to events — "When X happens, do Y" — use this command set (that is, the 'Y' is always a command from this set).
* Serial commands over the RS-485 port use this command set.

This document centers on the set of commands most useful to the ST9x integration effort, without getting into the weeds of unrelated commands. This Introduction attempts to convey the important architectural aspects of the AssetPack that help make the commands sensible. Foremost in this is the definition of key terms:

* a **Mode** describes the behavior of the unit: when it sends a report, when it sends a mailbox check, how it uses its sensors, how it uses the GPS, and most importantly, what Happenings it responds to and how. There are 14 Modes available on the unit, plus two (0 and 1) that are reserved.
* a **Happening** is an event that happens on the unit: motion starting or stopping, a sensor crossing a limit, the power state changing, geofence boundary crossing, anything we might want to respond to. There are 255 Happenings, and each Mode can decide which of those it responds to, and how: change to a different Mode? Send a message? Create a geofence? Any command can be used in a given Mode's response to a given Happening.
* a **Message** is a sequence of bytes sent over a backhaul (satellite, cellular, or neighborhood / WiFi). Each Message has a message type that lets the server know what the contents mean. There are 16 message types available; the meanings of 3 of them are fixed: 10 Mode Change, 14 Housekeeping, and 15 Exception. The rest have default meanings, but ultimately are application-defined.
* a **Sensor** is a source of numeric information: either a value, -32767 to +32767, that is read at a given time; or, the time that a sensor changed state. The former is referred to as an 'analog sensor', and the latter, a 'digital sensor'. Analog sensors can be compared against limits, and crossing a limit boundary results in a Happening that can be responded to.
* **Geofences** are geographic boundaries that define a Red zone, Green zone, and optional Yellow zone. The "Red" zone is, generally, the important place: it's the reason you made this geofence; if the unit goes into a red zone, you want to know about it. It doesn't mean it's bad. It just means it's the important side of the boundary. This is exemplified by the fact that the "Yellow" zone, if used, is the stripe of area within some distance of the boundary *approaching* the red zone: that is, "Yellow" means "Heads up, you're getting close to Red". The "Green" zone, then, is everyplace that isn't Red or Yellow.

For the ST9x integration, **Messages** are our primary concern. The ST9x will send commands over the RS485 port to ask the AssetPack to queue a message for communication over the satellite backhaul. This calls for some detailed awareness of how the AssetPack messaging subsystem works.

The AssetPack maintains a *message queue*, which is exactly what you would imagine: a list of messages in the order they were created, that get pushed out the door over a data backhaul whenever there's an opportunity to do so. What this means, is that information gets cleanly backfilled. If you were driving through a mountain pass and had trouble getting messages out, when you finally do get a connection, you will get out *all the messages that were queued up as you went through the pass*, and the information about where you were will be completely filled in.

Alright, so once a message is queued, it will eventually find its way out. What does queuing a message mean?

When the unit decides there's a message to send — a regular report, a sensor alert — it allocates space for it on the queue, and starts a three-step process:

1. INITIALIZATION: fill in the message type, maybe the extra nybble under the message type, and any custom initialization. When all the initialization steps have been fulfilled, the message proceeds to POPULATION.
2. POPULATION: gather and fill in the message contents. This is typically the GPS location, with some number of attempts to gather that location if the first one fails; geofence information; and any custom/sensor data to populate. When all the population steps have been fulfilled, the message proceeds to DISPATCH.
3. DISPATCH: outlink the message. There is a separate process that attempts to outlink using a neighborhood (WiFi, etc) link if available; failing back to a cellular link if available; failing back to a satellite link. Each message, though, can declare which backhauls it wants to use: some high-volume messages should only use neighborhood or cellular; some high-priority messages should skip straight to the satellite link. Each message in the DISPATCH state registers its desire for its applicable outlinks, and when an outlink starts, can fill in the last of its information such as battery and temperature levels, and any custom information. When a message is successfully outlinked, or in the rare case that it runs out of room in the queue, a message can be archived to the SPI Flash.

Each message is in one of these states at any given time. Only when it is completely done with a state, does it proceed to the next one. The message queue can always be assumed to have some mixture of messages that are initializing, populating, and waiting to be outlinked, all sharing the same queue.

INITIALIZE

More to do?

Nothing more

POPULATE

More to add?

Nothing more

DISPATCH

Not sent yet?

Sent

*Archive to SPI Flash*

When messages are dispatched, all of the messages that are in the DISPATCH state and which want to use whichever outlink has become available, are concatenated and wrapped into a transmission envelope. This envelope allows each message to be separated back out from the byte stream sent, and establishes the timestamps for each of the messages included. If the send succeeds, the sent messages are archived and their space in the queue freed up. If the send does not succeed, the AssetPack continues with its retry cycle, which is optimized for the backhaul being attempted. Note that whenever a new message enters the DISPATCH state, the retry wait time is short-circuited, and a new transmission attempt is immediately made.

On the chance that a series of long messages has been queued, or the AssetPack has been unable to communicate for an extended period of time, the queue may fill. Generally speaking, the oldest messages are kicked off the queue at this point. (There are provisions both for self-sacrificial messages — “I’m unimportant, eliminate me first” — and vital messages which can be stored in nonvolatile memory for later over-the-air retrieval, but it is simpler to regard the queue as one which kicks off old messages to make room for new ones if it needs to.) The yi command shows how much space remains in the queue.

**AssetPack Configuration for ST9x**

The AssetPack units that AssetLink is sending to SkyBitz/Ametek will have the following configuration:

* Unit activates in Mode 8 (“Timed Reporting”)
  + It sends a daily report (Message Type 13) on its own, independent of the ST9x
  + It performs a mailbox check (check for incoming messages) every hour
  + ANA\_DIO1 is configured as a digital change detect input
    - its debouncing time is 50ms
    - its calm time (where it will ignore changes after a debounced transition) is 2s
  + When ANA\_DIO1 transitions to 1:
    - external power is turned on
    - a timer is set for 2 minutes hence to turn off external power
  + When ANA\_DIO1 transitions to 0:
    - the timer to turn off external power is erased
    - external power is turned off
  + When battery voltage drops below 7.6V:
    - unit goes to Mode 9
* When in Mode 9 (“Timed Reporting, Power Save”)
  + It sends a daily report (Message Type 9) on its own, independent of the ST9x
  + It performs a mailbox check every 12 hours
  + ANA\_DIO1 is configured as above
  + When ANA\_DIO1 transitions to 1:
    - external power is turned on
    - a timer is set for 1 minute hence to turn off external power
  + When ANA\_DIO1 transitions to 0:
    - the timer to turn off external power is erased
    - external power is turned off
  + When battery voltage rises above 7.8V:
    - unit goes to Mode 8
* The RS485 driver is kept on, not powered down
  + The UART driver is turned off after 1 minute, however, so the ST9x should send a newline (0x0A) character periodically until it receives a prompt (“> “)
  + The UART parameters are 9600,n,8,1
* Messages 10, 14, and 15 are always fixed; messages 9 and 13 are daily reports as shown above; messages 0, 1, and 2 are reserved.
* The remaining message types are available for ST9x use, and are configured to accept up to 68 bytes (136 hex characters) via the “Message queue” commands:
  + Messages 8, 11, and 12 send the AssetPack’s status and location, followed by any bytes sent by the ST9x. Note that Message 11 is the default “Requested” message in response to a “Report ASAP” command from the AssetLink DeviceManager.
  + Messages 3 through 7 have no default contents and are intended to solely convey information as commanded by the ST9x.
  + Note that the “Message append” functionality can be used to send messages with more than 68 bytes, but please remain at or below 240 bytes total.

When commanding over the serial port, note the following special characters:

* LF (0x0A), CR (0x0D), or CRLF (0x0D0A), are accepted as line terminators. LF is preferred.
* ^C (0x03) cancels the current command.
* BKSP (0x08) backspaces.
* SHIFTIN (0x0F) is an escape character: whatever character follows it, is entered literally into the command buffer, without being interpreted as any of the above.

If the command buffer hits 144 characters, it the AssetPack will reject the command and emit a new prompt.

**Commanding Concepts**

There are some recurring themes in the river of commands to come, that it is worthwhile to cover once up front.

First is number terminology: let's say we're conveying the number 42. That's a decimal 42. If that is how we are to enter it in the command (numeral '4' followed by numeral '2'), the command will refer to that parameter as "decimal". If however the parameter is referred to as a "hex number", then 42 decimal = 2A hexadecimal, and we would type 2A or 2a. Finally, if the parameter is "raw binary", then that number 42 is actually a single byte. 42 decimal is an ASCII \* (asterisk), so the raw binary representation of 42 is \*. Most raw binary numbers are not printable or directly typable; raw binary is used for data compactness when a computer is trying to send a lot of information quickly to the AssetPack.

Briefly: 42 decimal = 2a hex = \* raw binary.

Speaking of numbers, there are several parameters that can only take on the values 0-15: mode numbers, message types, and sensor numbers. (Sensor numbers are actually typically 0-11, but they *definitely* can't go beyond 0-15.) These values are commonly expressed in three ways:

Decimal: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Hex: 0 1 2 3 4 5 6 7 8 9 a b c d e f

SHIFT-num: 0 1 2 3 4 5 6 7 8 9 ) ! @ # $ %

That last one is obviously the weird one. It was ironically intended to be easy to remember: to type a value 10 and above, hold SHIFT and press the last digit. SHIFT-0, for 10, is a close-paren. SHIFT-1, for 11, is an exclamation point. SHIFT-5, for 15, is a percent. I should have realized that "a through f" was just as easy to remember, but I didn't, and it became part of the interface, so here we are.

Next is command safety. A number of the commands in this document can be *made* dangerous. Most of the SPI Flash is just mass storage, and can be erased and written to with ease. But if a SPIF command goes and erases information critical to the operation of a given Mode, that can have serious consequences. Similarly, there are commands that are distractions to this document and therefore not included (like “T: Set Time”, which is done automatically via the GPS) but which nevertheless exist and might be inadvertently called.

If the AssetPack seems to have gotten a command that is making it behave in a weird way, carefully execute the following command:

!!!sudo:^Qj040a>>

**Importantly,** that ^Q is a genuine control-Q character (0x11).

This is a unit-reset command. It will address any problem that ultimately resides in RAM, but not any problem that resides in Flash (processor or SPI Flash). If the problem is resolved, you can execute

Ge 60000

to confirm to the unit that your problem is resolved. Alternately, to revert all the way to factory-original settings, enter the reset command (!!!sudo:^Qj040a>>) 9 times in a row. This will reset the AssetPack repeatedly, and ultimately engage a failsafe that reverts the unit to factory-original state, where it does not make any assumptions that any subsystems or memory spaces are already configured.

In the command definitions below, some conventions are employed:

* *<parameters>* are shown in italics, enclosed in angle brackets
* [optional] parts of the command are shown enclosed in square brackets
* **bold** is used to show an actual command as it might be used in practice
* spaces are meaningful! If a command shows a space, then one must be used. If a command shows no space, then none must be used.
* case is meaningful! Capital-M command "Mode state" is very different than lowercase-m command "Monitor". The only exception is hex values, which can tolerate a-f or A-F.

**Summary Table**

This table summarizes the APn commands described in this document. Click on the "Command Name" to jump to the detailed description of that command. Some commands have several variants in their detailed command descriptions; the link goes to the first (and generally most commonly used) one. Some commands are listed under different categories, because they apply to both, but the detailed description is only under one category, and all the links will bring you to that one.

|  |  |  |  |
| --- | --- | --- | --- |
| Prefix | Command Name | sudo? | Description |
| **Messaging Commands** | | |  |
| y[l|d|D| ] | [Message queue](#yl) |  | Send a message |
| y[p|P] | [Message append](#yp) |  | Append to a queued message |
| yx | [Mailbox check](#yx) |  | Check for incoming messages |
| yi | [Message list](#yi) |  | List all the messages currently in the message queue |
| ys | [Message status](#ys) |  | Show the state and contents of a queued message |
| ya | [Message abort](#ya) |  | Remove a message from the queue |
| yt | [Message config](#yt) |  | Show what a given message type is configured to send |
| yu | [Use backhauls](#yu) |  | Define which backhauls the unit should use vs. avoid |
| y1 | [Stay active](#y1) |  | Leave a backhaul active instead of power-cycling it |
| y0 | [Stop stay active](#y0) |  | Resume power-cycling of backhauls |
|  |  |  |  |
| **Mode Commands** | | |  |
| d | [Change mode](#d) |  | Go to a new operating mode, including power settings |
| M | [Mode state](#bigm) |  | Show operating mode configuration |
|  |  |  |  |
| **Information Commands** | | |  |
| v | [Version](#v) |  | Show unit firmware version |
| t | [Time](#t) |  | Show unit's timekeeping |
| V | [Battery voltage](#bigv) |  | Show current battery rail voltage |
| I | [System state](#bigi) |  | Show important system-wide parameters |
| M | [Mode state](#bigm) |  | Show operating mode configuration |
| 9 | [Comstats](#nine) |  | Show communication statistics |
| g | [GPS](#g) |  | Show or kick off a GPS acquisition to a given precision |
| ys | [Message status](#ys) |  | Show the state and contents of a queued message |
| yt | [Message config](#yt) |  | Show what a given message type is configured to send |
| E | [Error log](#bige) |  | Show the contents of the error log |
| z | [Sensor values](#z) |  | Show values and statistics recorded from sensors |
| G[r|R|^R] | [SPIF Read](#biggr) |  | Read contents of SPI Flash |
| G[h|H] | [SPIF Happening](#biggh) |  | Read out Mode-specific happenings from SPI Flash |
| GI | [SPIF Save Inits](#biggbigi) |  | Persist RAM-based parameters to SPI Flash in case of reset |
| Gi | [SPIF Unit Identity](#biggi) |  | Read out the unit’s serial number(s) |
| Gb | [SPIF buffers](#biggb) |  | Get the current memory write locations for buffers in SPI Flash |
| L | [Thermal limits](#bigl) |  | Read or Set temperature limits (also read temperature) |
|  |  |  |  |
| **SPI Flash Commands** | | |  |
| G[r|R|^R] | [SPIF Read](#biggr) |  | Read contents of SPI Flash |
| Ge | [SPIF Erase](#bigge) |  | Erase a 4kB page of SPI Flash |
| G[f|w|a|F|^F|^W|^A] | [SPIF Write](#biggf) |  | Write to or alter SPI Flash |
| GM | [SPIF Modbus](#biggbigm) |  | Write a Modbus command to be used during sensor gathering |
| G[h|H] | [SPIF Happening](#biggh) |  | Read out Mode-specific happenings from SPI Flash |
| Gi | [SPIF Unit Identity](#biggi) |  | Read out the unit’s serial number(s) |
| Gb | [SPIF Buffers](#biggb) |  | Get the current memory write locations for buffers in SPI Flash |
| Gc | [SPIF Checksum](#biggc) |  | Perform a verification check on a region of SPI Flash |
| Gs | [SPIF Self](#biggs) |  | Copy from SPI Flash to SPI Flash |
|  |  |  |  |
| **I/O Commands** | | |  |
| z | [Sensor values](#z) |  | Show values and statistics recorded from sensors |
| Z | [Sensor gather](#bigz) |  | Read one or all sensors |
| GM | [SPIF Modbus](#biggbigm) |  | Write a Modbus command to be used during sensor gathering |
| b | [Buzz](#b) |  | Make a tone from the buzzer |
| q | [Quiet](#q) |  | Shut down the commanding port for a period of time |
|  |  |  |  |
| **Command Envelopes** | | |  |
| # | [Parity check](#hash) |  | Perform a validation check before executing the command |
| ] | [Response limit](#closebracket) |  | Only send a portion of the command's response |
| + | [Do Later](#plus) |  | Execute this command at a later time |
| @ | [Run from SPIF](#at) |  | Run commands already loaded in SPI Flash |
|  |  |  |  |

**Messaging Commands**

The Messaging subsystem maintains a queue of messages that are waiting to be populated and/or offloaded via a backhaul. This means that there are two numbers associated with messaging: the message *types*, 0-15; and an *index* into the message queue, 0-255. While they are both numbers and both related to messaging, they refer to entirely different things. A message *type* has a fixed meaning: message type 10 (hex A), for example, is a message about a change in mode. When the mode changes, a message of type 10 will be added to the queue and get some index — let's say 29. That index is just a handle to reference *that particular mode change message* as it is populated and then sent out the door.

Some commands below deal in message types (*<msgtype>*), which are expressed as a single hex character (0-9,a-f); others deal in message queue indices (*<msgidx>*), expressed as a decimal number (0-255). The most common command is **yl*<msgtype>***: queue up a new message of this type, and let me know the *<msgidx>* it got.

yl*<msgtype>*[*<params>*][ *<add'l bytes...>*] Message queue, normal

Send a message of type *<msgtype>* (0-9,a-f), with normal behavior. If the message is configured to send a GPS location, it does. If the message is configured to send sensor data, it does.

If there is a space after the initial part of the command, all characters following it are parsed as a hexadecimal byte stream to be appended to the message. Generally speaking, message type *<msgtype>* must be configured to have a variable length, or the receiving systems will not recognize the *<add’l bytes>*.

Note that a given message must be already configured to accept arbitrary *<add’l bytes>* (specifically by sending an extra byte giving the message length), or it will be misparsed at the DeviceManager.

Esoteric: If there is more text immediately after the *<msgtype>*, the parser will read a hexadecimal number *<params>* that is fed into the msgStart() function. Its only use at this point is that if bit 0x10 in *<params>* is set, then the bottom nybble of *<params>* is injected as the “extra” nybble in the message.

yd*<firstbyte>*[ *<add'l bytes...>*] Message queue, manual

Send a message with entirely manual contents. *<firstbyte>* is a hexadecimal byte. Its upper nybble is the message type, the lower nybble is the “extra” nybble sent after it. Then there is a space, then a hexadecimal stream forming the entirety of the message contents. Generally speaking, the message type used must be configured to have a variable length, or the receiving systems will be confused by the nonstandard length and contents of the message.

yD*<firstbyte>*[ *<add'l bytes...>*] Message queue, manual, for append

Same as yd above, but sets a flag that freezes the message in the init phase so additional bytes may be added via the yp and yP commands. The response from the unit

Allocated msgtag 6

gives the message queue index, or tag, to be use by the subsequent yp and yP commands. WARNING: unavailable in versions < 1035.

yp*<msgidx>* *<add'l bytes...>* Message append

Append *<add’l bytes>* (a string of hex) to the message with tag *<msgidx>*. WARNING: unavailable in versions < 1035.

yP*<msgidx>*[ *<add'l bytes...>*] Message append and send

Optionally append *<add’l bytes>* (a string of hex) to the message with tag *<msgidx>*, then release the hold on the message and allow it to populate and send. WARNING: unavailable in versions < 1035.

y *<firstbyte>,<init>*,*<populate>*,*<dispose>*,*<custom>*,  
*<nmsgbytes>*,*<standard length>*,*<status>*,*<timestamp>*,  
[ *<add'l bytes...>*] Message queue, fully manual

Esoteric: manually fill all the parameters in the message structure and send that message. The first four parameters are hexadecimal, the last four are decimal. If the comma-separated list ends early, the remaining parameters are left as the default values for the given message type (defined by the upper nybble of *<firstbyte>*).

yx Mailbox checkin

Perform a mailbox checkin, querying the backhaul(s) for any messages being sent to the unit. Will also cause a retry cycle for currently pending messages.

yi Message list

List all the messages currently in the message queue. Example:

> **yi**

Messages in queue:

38: 85 @ 1aec @ 1495568421, 00 00000000 6080 01 8

40: 9e @ 1b1c @ 1495591875, 00 00000000 6080 01 8

41: 0b @ 1b34 @ 1495629858, 00 00000000 6080 01 8

42: 0b @ 1b4c @ 1495629858, 00 00002300 6280 00 8

top@ 1b20 wrt@ 1b20 (1326B spare)

For each line representing a message, the first number is the message queue index, or “tag”. As you can see, they will be grouped but not always be consecutive: message 39 apparently got dequeued somewhere along the line. The next number is the first byte of the message, in hex. The most important thing there is the first nybble (4 bits, the first character): that's the message type. So there is a message of type 8 (high priority regular report)[[1]](#footnote-1), type 9 (low priority regular report), and two of type 0 (panic / alert). The four-character hex number is where in memory the message starts; the long number starting with 149 is the timestamp on that message.

The next three values are the initialization flags, population flags, and disposal flags. The details of them are esoteric to the code but the most important thing is: Which ones are zero? A message only goes to the next phase (init to populate, populate to dispose, dispose to archive) when the previous phases' flags have all been zeroed, indicating the message is done with that phase. All four of these messages are done with their initialization phase. The first three are done with their population phase and are currently trying to be offloaded: the status of 01 in the second-to-last value shows that a backhaul is connected and the message on that line is being sent. (If the send fails, those status values will return to 00 until another backhaul connection is made.) The last message is still waiting to be populated with data (usually a GPS fix), after which it will be ready for offload. It is not being sent now (00 in the second-to-last position) because it isn’t fully populated.

The final number is the size of the message in bytes. Note this does not count any message envelopes, such as the message size or timestamp or encryption / validation data, which are included immediately before transmission.

The bottom line is a summary of the message queue usage: the address of the top of the message queue, the write pointer into the message queue, and the number of bytes available in the message queue.

ys[*<msgidx>*] Message status

Show the status of the message queue, or a particular message within it. Remember that *<msgidx>* is not a message type, but a (decimal) index into the message queue, called a “tag”. Examples:

> **ys**

mout 0000 *message outlink engine status; see below*

orrd 00 *Esoteric: backhaul override, as set by yu or y1 command*

mask 009f *Esoteric: backhaul status mask*

tag0 1 *lowest message queue index in use*

tagm 5 *next message queue index to allocate*

writ 1b20 *write pointer into message queue*

nbyt 37 *number of two-byte words used in message queue*

OK1

>

The message outlink engine status is a bitfield with the following meanings (values in hex):

0000 outlink is idle

0001 a backhaul modem is powered

0002 a backhaul modem is connected to its network

0003 outlink is waiting to retry a backhaul

0004 this communication is only a mailbox check

0010 an important message has just shown up: try to send it, even if we were waiting

0020 currently trying a neighborhood (WiFi, LoRa, etc) backhaul

0040 currently trying a cellular backhaul

0080 currently trying a satellite backhaul

0100 backhaul modem is to stay active, not power-cycle

0400 cellular backhaul should stay connected to its network independent of the messaging system

1000 something just happened that needs a modem power cycle to recover, even if the modem is otherwise supposed to stay active

0200 Esoteric: encryption state

0800 the shallow message buffer[[2]](#footnote-2) just got its first message

2000 the DeviceManager has been informed that the shallow message buffer is filling

To look at a particular message in the queue, use its message queue index, or tag:

> **ys42**

Message #42:

nbyt 12

init 0

fpop 2300

disp 6280

stat 0

lens 8

cust 0

ntime 1495629858

data: 3b 7f ff 00 00 00 00 00 11 22 33 44

OK1

>

msgidx 42 has 8 bytes of message data: 8 standard (lens), containing GPS location and other fixed data, plus 4 custom-population bytes such as sensor data (nbyt=lens+4). It is done initializing, but still needs to finish populating and then dispose / offload the message. Its status of 0 tells us it is not currently sending over a backhaul (good, because it is still populating the message data). The cust field would be for custom coded applications. The data bytes as currently populated are shown: it is a message of type 3, with an “extra” nybble of b (11), and the rest of the bytes are still being filled in.

ya*<msgidx>* Message abort

Dequeue the message at *<msgidx>* in the queue.

yt*<msgtype>* Message config

Show the configuration of the given message type: that is, when a message of this type is queued, show what it will do. Example:

> **ytd**

MSGTYPE d

dlen 8

init 00

pop 3310

disp 42e0

cus0 00000000

cus1 00000000

INST0: 01

INST1: 00

INST2: 01

INST3: 00

OK1

>

Message type 13 (0xD) has a default length of 8 bytes. It has no special instructions on initialization, but will try up to 3 times to get a GPS fix (the first '3' after 'pop'), does not try to populate sensor information (the '0' after 'pop'), and is happy going out any available backhaul (the 'e' after 'disp'). Its sensor population instructions say to start at instruction 1 and end at instruction 0, which is another way of saying "Don't populate any sensor information". (As you can tell, this is a largely esoteric and code-specific set of information.)

yu*<backhaul>* Use backhauls

Override the unit’s own knowledge of what modems it has available, substituting the hexadecimal value *<backhaul>*, which is the logical OR of:

8: satellite (Iridium or Globalstar)

4: cellular

2: neighborhood (WiFi or other LAN)

Somewhat obviously, this command can only turn *off* modems, relative to “all that you have”; the command will ignore (zero out) any bits that represent modems it knows are not physically available.

A *<backhaul>* of zero will restore the default value of “all modems physically available”. Also, if the unit enters a “suitcase condition” — it detects that it cannot communicate for several days — it will override this command and allow communication over all backhauls.

y1*<backhaul>* Stay active

Turn on the *<backhaul>* modem and leave it on. No other backhaul will be tried. *<backhaul>* can be:

8: satellite (Iridium or Globalstar)

4: cellular

2: neighborhood (WiFi or other LAN)

If the unit enters a “suitcase condition” — it detects that it cannot communicate for several days — it will override this command and allow communication over all backhauls.

y1*<backhaul>* *<timeout> <mailbox\_check\_pace>* Stay active (auto)

Same as the previous command, but adds automatic behavior for the unit to perform. *<timeout>*, in seconds up to 65535 (0 means no timeout), is the time until the unit should issue itself a "Stop staying active" command. *<mailbox\_check\_pace>*, in seconds up to 65535 (0 means no mailbox checks), is the pace at which the unit should send a mailbox check, independent of the mailbox checks driven by the mode schedule.

y0 Stop staying active

Allow the backhaul turned on by a "Stay active" command, to turn back off at the next opportunity. Normal backhaul usage and power cycling will then resume.

**Mode Commands**

dn*<modenum>* Change mode

Change to mode *<modenum>* (0-9,a-f). *<modenum>* is actually parsed as a full hexadecimal number, with the following flags in the upper bits:

0x20 Sing the mode-start song

0x40 Don't send a mode change message

0x80 Keep the 'mode start time' unchanged (so the schedule doesn't restart from the beginning)

So the command dne will go into mode 14 (0xE) and send a mode-change message, but dnce (0xC = 8+4) will go into mode 14 while keeping the current mode-start time and not sending a message.

Important: this is probably the most dangerous non-sudo command. It can send the unit into an undefined mode, or even a defined mode with undesired behavior. When executing this command, make absolutely sure it is aiming at the right target *<modenum>*. As one double-check, the last character in this command should never be zero, as Mode 0 is effectively a radio-silent mode.

dinventory Change mode to Inventory

Go into Inventory mode. While this is not a sudo command (because it cannot permanently brick the unit), it will send the unit offline.

da Change mode to Active

Go into whatever mode is this unit’s usual starting mode, leaving Inventory if needed. Generally equivalent to moving the magnet to “ON”.

dA Explicitly activate

Whether we were in Inventory or not, go through the full leaving-Inventory startup sequence.

dp*<power level>* Change power level

Change the power level to the hexadecimal value *<power level>*. The lower three bits define the power level itself:

1. Desperate (Power Recovery)
2. Power Save
3. Normal
4. Full

The upper nybble has flags:

0x10 Sticky: the code should not subsequently change the power mode

0x20 Suitcase: the power level is because of sky view, not battery level

0x80 Inventory: the unit is in Inventory mode

The code performs all actions that result from the power level changing.

df*<power level>* Force power level

Change the power level to the hexadecimal value *<power level>*, defined the same as above. This command only sets the power level status, not the operating mode. That is, the variable representing the power level is changed, and any future decisions based on that level will be affected; but this command does not cause a Happening of "Power level changed" nor any resultant behaviors.

dh[f] Start heartbeat

The periodic heartbeat that drives the mode schedule, is restarted. This can be done for synchronization (the first heartbeat will hit an exact amount of time — defined by the system parameter [fast\_]heartbeat\_start\_sec — from the issuance of this command), or to change which kind of heartbeat to use: normal (every 15 minutes, the mode's schedule spans 24 hours), or if the f is included, in the command, fibrillating (every 37.5 seconds, the mode's schedule spans 1 hour).

M[*<modechar>*] Mode state

Show the configuration of the current operating mode, or of mode *<modechar>* (0-9, SHIFT 0-5) [[3]](#footnote-3). Example:

> **M)**

Mode 10 (now 2 started 2750 ref 2750)

power: 4

flags: 280

ad1: 40

ad2: 40

sense@: 0

settle: 800

gps@: 15

cust: 0,0

geofA: 0000

geofB: 0000

geofC: 0000

Schedule: 000000000000c00000000000000000000000400000000000

Triggers: 3, 5, 8, 10, 16, 17, 20, 40, 41, 44,

OK1

The configuration of Mode 10 (SHIFT-0, which is a right paren) is shown. The unit notes that it is currently in Mode 2, and that it entered mode 2 at timestamp 2750, with a mode-start time also of 2750 (these numbers can be different if a mode was entered with flag 0x80, "Keep the mode-start time", [see dn command above](#d)). The current power level is 0x04, with the bits defined as [in the dp command above](#dp): in this case, 0x4 Full power available.

The remaining lines show the configuration for Mode 10. Its flags, use of ANA\_DIO1 and 2, rate of reading sensors, sensor settling time, GPS periodic check time, and geofence check flags, are listed; the individual meanings of all of these are esoteric to the code. The schedule is a hex stream where every pair of bits shows what to do with each heartbeat:

0b00 Do nothing

0b01 Perform a mailbox (incoming-message) check

0b10 Do some custom action

0b11 Send a normal report for this mode

The "Triggers" list is the set of all [Happenings](#apphappen) that this mode responds to.

**Information Commands**

v Version

Show details about the firmware version running. Example:

> **v**

VERSION NUMBER: . 732:e104708c92b5

COMPILATION DATE: . 2017-11-10T18:35:33Z

LATER UPDATES: . Original

OK1

t Time

Show details about the unit's timekeeping. Example:

> **t**

Real: 480.158

T1: 5331

@T1=0: 479.995

@start: 0.000

@adj: 0.000

delta: 0.000

tick: 30.518

OK1

The "Real:" value is the Unix time (seconds since 1 Jan 1970) the unit thinks it is, as updated by GPS. The low value shown in this example shows the unit has not yet gotten a GPS fix. The remaining values are details about the unit's time calibration efforts.

Esoteric: if there is any value after the t, the unit sends an abbreviated version of the time information. This is helpful to verify an OTA command to the unit: if the last command sent in a stream OTA is t*Number*, the unit will send back this t*Number* and the time it was executed. If the server receives this reply, it knows both that all the preceding commands were executed successfully, and at what time they completed.

V Battery voltage

Get an immediate reading of the battery voltage, in milliVolts. Note that for the AP3, this requires momentarily turning the 9602 on.

I System state

Show important parameters within the sys structure. Not all system parameters are shown here, just the ones that are either most impactful to the user (like power limits), or most likely to be tweaked (like GPS radii). Example:

> **I**

System config

valid=9a04 *9a04 means this config is valid*

version=32 *interface version used by this unit*

behav=0021 *hex value with flags defining overall behavior*

start=2 *mode to start in, when the unit is turned on*

hb=9000 *# seconds for the normal heartbeat*

fhb=375 *# tenths of seconds for the fibrillating heartbeat*

gmo0rad=200 *GPS fix must be within this many meters of previous fix to mean Motion Stop*

gmo1rad=200 *GPS fix must be further than this many meters of previous fix to mean Motion Start*

gmointer=200 *GPS fixes must be > this # seconds apart to be compared for motion*

gmomarg=10 *meters of margin to try to minimize overreaction to being near a boundary*

uartwake=60 *RS485 command port is left fully powered up for this # seconds*

llwd=630 *Long Leash Watchdog # seconds; after this, RS485 is fully idled*

sendone=15 *if we have spent this many seconds reading sensors, we declare we’re done*

sersens@68000 200 *serial sensor instructions are @this SPIF address; replies time out in this many ms*

power: 8050 -> 7600 -> 6800, 6900 -> 7700 -> 8150 *see below*

suitcase: 55 -> 111 *see below*

cust: 0,0,0,0,0,0,0,0,0,0, *parameters for custom code*

geofA: 0 0 *# minutes meaning we’ve been inside / outside a fence for long time, for geofence set A*

geofB: 0 0 *# minutes meaning we’ve been inside / outside a fence for long time, for geofence set B*

geofC: 0 0 *# minutes meaning we’ve been inside / outside a fence for long time, for geofence set C*

geofI: 0 0 *# minutes meaning we’ve been inside / outside a fence for long time, for geofence set I*

OK1

The power and suitcase lines are for power management. The power numbers are all in milliVots. The first triplet is the descending power level thresholds: to go from Full to Normal; then Normal to Power Save; then Power Save to Power Recovery. The second triplet is the ascending power level thresholds: Power Recovery to Power Save; Power Save to Normal; and Normal to Full. The suitcase numbers are in hours: first, the hours until "cannot see the sky" sends us into Power Save; second, the hours until it sends us into Power Recovery.

M[*<modechar>*] Mode state

[See above under "Mode Commands".](#bigm)

9 Comstats (all)

Display statistics taken during each of the last four backhaul communication attempts. These four attempts are a rolling list labeled 0 through 3; this command lists them all, starting with the most recent and going back in time. In the example below, we define all the fields for the first (most recent) data block, then highlight in the subsequent data blocks those fields that have changed in interesting ways.

> **9**

COMSTATS #0 (current 0) *Slot 0 has the most recent comms attempt*

GPS: *This was the most recent GPS fix at the time of this comms attempt*

pos 38.922538,-77.039623 *Latitude,Longitude*

time 54068.000 *Time field from the $GPGGA GPS sentence*

hdop 0.850 *Horizontal Dilution of Precision from the GPS*

alt 99.599 *Altitude in meters from the GPS*

hght -33.500 *Height of geoid above WGS84 ellipsoid from the GPS*

when 1575471668 *Unix timestamp of when this fix was taken*

dgps 0 0 *DPGS time and station ID*

fixq 1 *Fix quality: 0 is invalid, 1 is GPS fix, 2 is DGPS fix*

nsat 9 *Number of GPS satellites used*

ttff 20 *Time to first fix, in seconds*

comst 1575471668.475 *Unix time when this communication attempt started*

locst 1575471601 *Unix time when the above GPS fix attempt started*

cmacq 7.38 *Time until network acquisition, in seconds*

cmdon 16.25 *Time until communication done, in seconds*

count 56 *Number of communication attempts total that this unit has tried (up to 65,535)*

mfcnt 56 *Number of communication attempts since this unit’s last manufacturing test (up to 65,535)*

exchg 0 *Result code for the communication attempt as reported by the modem[[4]](#footnote-4)*

r@acq 5 *RSSI at network acquisition as reported by the modem*

r@end 5 *RSSI at communication done as reported by the modem*

flags 80 *Which backhaul was tried: 80=satellite, 40=cellular, 20=neighborhood*

COMSTATS #3 (current 0) *Slot 3 has the second most recent comms attempt*

GPS:

pos 38.922723,-77.039546

time 46903.000

hdop 1.210

alt 54.900

hght -33.500

when 1575464503

dgps 0 0

fixq 2

nsat 6

ttff 55

comst 1575464984.000

locst 1575464401

cmacq 5.41

cmdon 19.48

count 55

mfcnt 55

exchg 32 *This exchange failed: we connected to the network but could not get our message through*

r@acq 0 *And here’s why! Our RSSI was zero when we managed to connect to the network,*

r@end 0 *… and still zero when the modem reported the failure*

flags 80

COMSTATS #2 (current 0) *Slot 2 has the third most recent comms attempt*

GPS:

pos 38.922723,-77.039546

time 46903.000

hdop 1.210

alt 54.900

hght -33.500

when 1575464503 *The most recent successful GPS fix was the same for the comms attempts in slots 2 and 3*

dgps 0 0

fixq 2

nsat 6

ttff 55

comst 1575464503.396

locst 1575464401

cmacq -0.01 *A negative number here says that network acquisition never happened*

cmdon 30.35 *After 30 seconds waiting for network acquisition, we timed out*

count 54

mfcnt 54

exchg 65535 *There was no exchange attempted*

r@acq -1 *There was no RSSI at network acquisition because there was no network acquisition*

r@end 0 *Unsurprisingly, the RSSI for a network that we couldn’t find is zero*

flags 80

COMSTATS #1 (current 0)

GPS: *Slot 1 has the fourth most recent comms attempt*

pos 38.922818,-77.039801

time 39663.000

hdop 0.829

alt 100.800

hght -33.500

when 1575457262

dgps 0 0

fixq 1

nsat 9

ttff 15

comst 1575457740.339

locst 1575457201

cmacq 7.75

cmdon 26.16

count 53

mfcnt 53

exchg 32 *This exchange failed like the one in slot 3,*

r@acq 3 *… but with an RSSI of 3,[[5]](#footnote-5) implying that the signal was present but distorted*

r@end 3

flags 80

OK3

9c Comstats (current)

Same as above, but only shows the most recent entry.

9*<num>* Comstats (specific)

Same as above, but shows slot *<num>*. This is out of the range 0-3, where any of them may be the most recent.

g*<level>* GPS fix

Show all GPS information at the given location quality level, defined as:

LOC\_QUAL\_IMMEDIATE 0 *most recent GPS sentence received*

LOC\_QUAL\_QUICK 1 *GPS sentence just after it settles on a fix*

LOC\_QUAL\_NORMAL 2 *fix we have confidence in*

LOC\_QUAL\_ALMANAC 3 *fix from when we were trying to get a piece of the almanac*

LOC\_QUAL\_FULMANAC 4 *most recent first fix [FULMANAC actually means that we’re leaving*

*the GPS on as long as we can, so there is no actual fix associated]*

The output is the same as for the GPS section of the [9 command](#nine) above.

gg*<level>* GPS get fix

After showing the current fix at the given level, request a new fix at that level. If one is immediately available, it is shown. If not, the word "Reacq..." is shown, indicating the location subsystem is getting a new fix to the desired quality level. The same command can be sent multiple times with no ill effects, and when the response is no longer "Reacq..." but rather two copies of a GPS fix, you know that is an up-to-date fix.

*<level>* is actually parsed as a hex value, so it can include flags, most particularly LOC\_FORCE\_START (0x10), which will start a new acquisition even if a recent fix is available. Note that this can still be overridden by the location subsystem if it is in the middle of something like an almanac download.

gv GPS velocity

Show the most recent calculated unit course and speed. Speed is in meters per second, course is in degrees off North, +/-180. The AssetPack calculates this using a "least-squares curve fit" during the most recent LOC\_QUAL\_NORMAL fix, to get the best overall velocity vector without local variations (like rocking back and forth).

When this command is executed, the speed (in tenths of a meter per second) is copied into the last\_sensor\_reading variable. It can then be recorded as a sensor, if desired, for limit checking or statistics gathering. (Alternately, there is a mode setting which always copies the speed into Sensor 5.)

L[m] Thermal limits (show)

Shows the last temperature measured, the cold thermal limits, and the hot thermal limits. When a unit measures a temperature beyond its limits, it sends an alert over the air, then pushes the limit temporarily 2 degrees further. Example:

> **L**

Last: 61 (6.1C)

Cold: -400 (-40.0C)

C(now): -400 (-40.0C)

C(rst): -380 (-38.0C)

Hot: 700 (70.0C)

H(now): 700 (70.0C)

H(rst): 680 (68.0C)

OK1

"Last" is the last temperature measured. The thermal limit (Cold / Hot) is the initial one the temperature has to go past, to send an initial alert. The "now" limit is the limit the temperature has to go past, to send another alert ("Hey, still having a problem"). The "rst" limit is the temperature that needs to be retreated past, for the unit to consider itself back out of danger.

With the m flag, the unit takes a fresh temperature reading before this printout.

E[*<number>*] Error log

Show an entry in the unit's nonvolatile error log. If *<number>* is included and is positive, reads out that slot in the error log (0-95). If *<number>* is not included, reads out the most recent error.

Esoteric: If *<number>* is negative, reads out archived data that get stored to the upper byte of the processor's nonvolatile memory. This allows you to reach a little further back into the error log history.

Example:

> **E0** *get the error recorded in slot #0*

Current error pointer 0c20 (#1) *the most recent error is in slot #1*

Error #0 (8c00): type 0001 @ 0.000000 w/ 00080083 X ff *see below*

MM 0000 *major mode at the time of the error*

mm 0000 *minor mode at the time of the error*

PA 0001 *peripherals active at the time of the error*

BV ff08 *boot values at the time of the error*

crit 0008 *critical individual bits at the time of the error*

writ 0080 *message buffer write pointer at the time of the error*

pwrl 83 *power level at the time of the error*

msgo 00 *message-out state at the time of the error*

#evn 00 *number of scheduled events at the time of the error*

batt fa *most recently measured battery level at the time of the error*

tdel 0 *offset of calibrated time per T1 tick vs nominal, in units of 1/64us, at the time of the error*

OK1

As you can see, most of these values only make sense in the context of the details of the software, and are intended to capture the most likely indicators of a cause: what was active, was the timekeeping reasonable, did the backhaul get confused. These values are only meaningful when compared to how they are used in the code.

The top line, though, gives generally useful data: which slot is this (slot #0 as requested), where is it recorded in memory (8c00), what is the type of error ([as defined in std.h](#appstd); in the example above, error 0001 is FAULT\_RESTART, that the processor reset), when did it happen (in the above case, at 'time 0.000'), what extra information did the code want to record (FAULT\_RESTART records the processor's registers capturing the reason for the restart), and finally, how many times did this error repeat in a row. The 'ff' indicates that this error occurred once; if it repeated, a bit would be lost each time: 7f means the error repeated once, 3f twice, 00 means it repeated 15 times or more after the original event.

z Sensor values

[See below under "I/O Commands".](#z)

**SPI Flash Commands**

In the ST9x application, these commands may be used in future as a mailbox from the SkyBitz servers through the AssetPack to the ST9x. With an addition to the DeviceManager API specific to SkyBitz, the SkyBitz server could write to custom pages 8-23 however it wishes, and the ST9x could read this information (using the commands below) when it powers up.

By convention, the AssetPack SPI Flash or SPIF is organized as follows:

0x000000 Original program image from the factory

0x020000 Program image A, the most recently uploaded image

0x040000 Program image B, the most recently configured image

0x060000 Special bootloader instructions

0x061000 Responses to Happenings for each Mode

0x065000 Manufacturing test data

0x067000 Persistent initialization values, e.g. for sensor limits

0x068000 Custom page 0: sersens/Modbus commands followed during regular sensor gathers

0x069000 Custom page 1: sersens/Modbus commands to be executed immediately

0x06A000 Custom pages 2-3: configuration change commands

0x06B000 Custom pages 4-7: configuration change history

0x070000 Custom pages 8-23: genuine application-specific use

0x080000 Scratchpads for data copying

0x0C0000 Shallow message storage

0x0C4000 Deep message storage

0x100000 Monitor mode internal record

0x200000 Geofences

0x3FFFFF Top of SPIF

The only absolute is the region from 0x000000 to 0x01FFFF, which is locked at manufacture time and the firmware will refuse to erase or write to it. This is to ensure that no matter what else happens, the bootloader always has a known-good copy of code to fall back on. In turn, the bootloader uses the first byte of the “Special bootloader instructions” page as a countdown: every time a reboot occurs, a 1 is stripped off this byte, from 0xFF to 0x7F to 0x3F and so on down to 0x01 and then 0x00. When this countdown hits zero, the bootloader knows something has gone terribly wrong that the operational code is resetting so much, and copies the original program image into the processor to try to recover. (The command Ge 60000 will reset this countdown.)

The flip side of this is that the rest of SPIF is readily accessible to all the below commands. This means it is easy to read, write, and manipulate this information; but it also means that it is easy to accidentally erase geofences, or make Happening responses nonsensical, or turn the other images to a garbled mess. By and large, it is useful to stick with the “custom page” shortcut described in the below commands, which at least guarantees that no information below custom page 0 can be accidentally overwritten.

Gr|R|^R[c] *<address>*/*<# bytes>* SPIF Read

Read out bytes from SPI Flash, starting at the given *<address>* and continuing for *<# bytes>*. The two parameters are separated by a slash. If the optional c is used, *<address>* is a decimal number indicating which SPIF custom page to use; otherwise, *<address>* is a hexadecimal address. *<# bytes>* is always decimal.

* r: print the starting address, then the bytes in space-delimited hex form. *<# bytes>* is limited to be no more than 64, to avoid spending too much time in the printout.
* R: print the address and data in the Intel Hex format. *<# bytes>* is limited to be no more than 64.
* ^R: print the bytes in raw binary form. There is no limit to *<# bytes>*, though an extraordinarily large number may take so much time that it will (correctly) trip a watchdog reset.

Ge[c] *<address>* SPIF Erase

Erase the 4096-byte page in SPI Flash that includes *<address>* (in hex). If the optional c is used, *<address>* is a decimal number indicating which SPIF custom page to use; otherwise, *<address>* is a hexadecimal address.

Gf|w[c] *<address>*/*<bytes...>* SPIF Write (text)

Write *<bytes>* into SPI Flash, starting at the given *<address>*. *<bytes>* are two-character hex values with no separation. If the optional c is used, *<address>* is a decimal number indicating which SPIF custom page to use; otherwise, *<address>* is a hexadecimal address. Note this command does not first erase the bytes it is about to write to: writing to an area of SPI Flash that has already been written to may have unintended results.

* f: start writing at the first blank area following *<address>*. A blank area is defined as 5 0xFFs in a row. The number of bytes to be written must not exceed 64; if it does, the whole command is considered invalid.
* w: start writing exactly at *<address>*. The number of bytes to be written must not exceed 64; if it does, the whole command is considered invalid.

G^F|^W[c|a[n]] *<address>*/*<bytes...>* SPIF Write (binary)

Same as the SPIF Write command above, but where *<bytes>* are raw binary data with the MSb inverted. That is, if you want to send the bytes FF, 00, 55, you'd send 7F, 80, D5 (again as raw binary bytes). This is because intervening interpreters might capture control characters, like ^C (0x03) or BKSP (0x08), but it is common to have those bytes (particularly NUL, 0x00) in data being written. Thus to avoid having to escape a large portion of the bytes being sent, a relatively small portion of the bytes (at and just above 0x80) need to be escaped.

If the n flag is included, the inversion is skipped. This is helpful for writing text into SPIF, particularly for serial commands. The funny syntax is just to put the characters in the right place:

*nothing* *<address>* is absolute hex; MSb is inverted

c *<address>* is a decimal SPIF custom page number; MSb is inverted

cn *<address>* is a decimal SPIF custom page number; MSb is not inverted

a *<address>* is absolute hex; MSb is inverted

an *<address>* is absolute hex; MSb is not inverted

GFc|a[n] *<address>*/*<command>* SPIF Write (command)

Convenience function for creating commands to be executed by a later [Run From SPIF](at). Assumes *<command>* is a text command to be appended to a series of them in SPI Flash. In this command, the optional n means "RespoNd", meaning the response to *<command>* should be queued for transmission over the backhaul. Here is a common usage:

Gec 22

GFc 22/!!!sudo:Cm2x 0 55d555d555d555d555d555d555d555d555d555d555d555d5>>

GFc 22/!!!sudo:Cm3x 0 00c000c000c000c000c000c000c000c000c000c000c000c0>>

GFcn 22/t5

@c22

This places commands to update the schedules for Modes 2 and 3 (responses not sent), followed by a command to get a timestamp (response sent to let us know the commands were all executed), into SPIF custom page 22, and then runs the updates atomically.

Ga|^A[c] *<address>*/*<bytes...>* SPIF Write (alter)

Change the given bytes in the SPI Flash to the given values, keeping all other bytes already in that SPI Flash page unchanged. This involves copying the whole page over to a temporary area, substituting *<bytes>* starting at *<address>*, erasing the original page, and copying the modified copy back. With a, *<bytes>* are hexadecimal text. With ^A, *<bytes>* are raw binary data. If the optional c is used, *<address>* is a decimal number indicating which SPIF custom page to use; otherwise, *<address>* is a hexadecimal address.

GMc *<page>*/*<fun> <addr> <reg> [<sensor> [<#regs>]]*

SPIF Modbus (read)

At the SPIF custom *<page>* (in decimal), append a Modbus sersens command with the following parameters, all in decimal:

* Modbus function number *<fun>*
  + 1: Read coil status
  + 2: Read input status
  + 3: Read holding registers
  + 4: Read input registers
* Modbus device address *<addr>*
* Modbus register number *<reg>*[[6]](#footnote-6)
* The sensor number *<sensor>* to read the first result into (4-11). A sensor number greater than 16[[7]](#footnote-7) will cause the read to occur but not be recorded as a sensor value (this is typically most useful to put the reading in the immediate last\_sensor\_reading variable for further manipulation). Default value is 98, which is a noticeably unusual value if it is misparsed and shows up in an unexpected place.
* The number of registers to read *<#regs>*, default 1.

This is a convenience function to parse the decimal values that go into a Modbus read command, convert them to binary for later transmission, wrap them in the sersens command envelope, and append them to a set of sersens commands being constructed.

Esoteric: the c is optional, as in all the other SPIF commands, but this command would rarely be used to an absolute address.

GMc *<page>*/*<fun> <addr> <reg> <value> [<sensor>]*

SPIF Modbus (write)

At the SPIF custom *<page>* (in decimal), append a Modbus sersens command with the following parameters, all in decimal:

* Modbus function number *<fun>*
  + 5: Write single register
  + 6: Write single coil
* Modbus device address *<addr>*
* Modbus register number *<reg>*[[8]](#footnote-8)
* Value to write to that register *<value>[[9]](#footnote-9)*
* The sensor number *<sensor>* to write the first response into (4-11), which should be an echo of the value written. A sensor number greater than 16 will cause the response to be ignored. Default value is 99, which is a noticeably unusual value if it is misparsed and shows up in an unexpected place.

This is a convenience function to parse the decimal values that go into a Modbus write command, convert them to binary for later transmission, wrap them in the sersens command envelope, and append them to a set of sersens commands being constructed.

Esoteric: the c is optional, as in all the other SPIF commands, but this command would rarely be used to an absolute address.

Gh *<modenum>* SPIF Happening (parsed)

Show all of the responses that the given *<modenum>* (0-15) has to all Happenings. Example:

> **Gh 3**

@061c00

3 5 : yl9

6 5 : dn2

7 5 : dn2

8 5 : dn5

10 5 : dn5

16 5 : yl7

17 5 : yl7

18 9 : Fs 40 6

20 e : yl7 : Fs 40 5

40 5 : yl7

41 5 : yl7

44 5 : yl7

OK1

These are the responses that Mode 3 has to any Happenings. The starting address of these responses is 0x061C00 (in SPI Flash). There are responses to Happenings 3, 6, 7, ... 40, 41, 44. After each Happening number, is the length of the response records to follow, in hex (note: this length value may have flag bits set, making the numbers appear large). After that length is all commands that are executed when that Happening is triggered, separated by colons.

GH *<modenum>* SPIF Happening (raw)

Show the region of SPI Flash memory that defines all of the responses that Mode *<modenum>* has to all Happenings. They are displayed as hex bytes and stop at the end of the responses with the byte FF.

Gc[c] *<address>*/*<length>* SPIF Checksum

The unit performs a CRC checksum, starting at *<address>* (in hex) and going for *<length>* (in decimal) bytes, and returns the result. The caller can perform the same checksum on what they expect this region of SPIF memory to contain to confirm or deny that the contents are as expected. The checksum calculation uses a 32-bit polynomial of 0x04C11DB7 ([CRC-32](https://en.wikipedia.org/wiki/Cyclic_redundancy_check)), initialized at 0xFFFFFFFF.

If the optional second c is used, *<address>* is a decimal number indicating which SPIF custom page to use; otherwise, *<address>* is a hexadecimal address.

Gc[c] *<address>*/*<length>*/*<expected>*[: *<command>*] SPIF Checksum Check

The unit performs a CRC checksum as above, but compares the result to *<expected>*, returning a value of +101 on success and -101 on failure. However, the real use of this command is to conditionally execute *<command>*: if and only if the checksum matches the *<expected>* value, the *<command>* is executed, and its return value is returned.

In this command, the optional extra c flag to change *<address>* to be a decimal SPIF custom page number is truly a useful flag, with both uses common: sometimes it is exactly what is wanted ("I just sent a bunch of commands to this custom page: check them before executing"), sometimes an absolute address is wanted ("I just sent a new firmware image: check it before loading it onto the processor").

Note that for consistency with the "Do Later" command, an explicit space before *<command>* is mandatory.

Gs *<spif\_read\_addr>*/*<spif\_write\_addr>*/*<length>* SPIF Self

Copy *<length>* (in decimal) bytes from the SPIF starting at *<spif\_read\_addr>* (in hex), into the SPIF starting at *<spif\_write\_addr>* (also in hex). As the SPIF write address hits a page-start address, that page is erased, otherwise it is not; *<spif\_write\_addr>* usually starts at a page start address (XXX000) for this reason.

Esoteric: the c flag to change *<spif\_read\_address>* to be a decimal SPIF\_CUSTOMS page number can be used, but that could be confusing, because *<spif\_write\_addr>* is still an absolute hex address.

GI SPIF Set Initialization Values

Some parameters, like sensor limits and the unit's ESN, are stored in RAM and can be lost in a processor reset. Issue this command to persist the current values to SPIF, where they can be reloaded in case of reset.

Gi SPIF Read Identities

Print out the modem ESN (aka IMEI aka CCID) information that was recorded during initial manufacturing.

Gb[0] SPIF Buffers

Print out the current SPIF write addresses for the message shallow store, followed by the message deep store. The message deep store is a record of all messages sent, attempted to be sent, and received, plus communications statistics for all comms attempts. It is intended to store thousands of messages in a circular buffer.

The shallow store, on the other hand, is a linear stretch of 16 kiloBytes, which a message can be configured to be stored into only if it is about to be pushed off the transmit queue but it is too important to lose. That is, if there is a particularly high-value message that should survive even if the AssetPack cannot communicate for an extended period of time — meaning that the data are highly valuable even if they are days or weeks old — that message can be configured to be saved in the message shallow store, where it can be read out by a command from the DeviceManager.

This command gives information on where the shallow store write buffer is, and therefore on how many such messages are waiting there to be read off. Once they are read off, the DeviceManager can use the optional 0 suffix to erase the message shallow store to make room for more such messages.

**I/O Commands**

These commands are not presently very relevant to the ST9x integration; but they may be useful in future, so they are included here.

The AssetPack processor has a great deal of important I/O, but most of that I/O — say, GPS interaction, or charging control — is baked into the base code and runs on its own. The real excitement is sensors — starting with ANA\_DIO1 and ANA\_DIO2, the general-purpose I/O that connect outside the box; plus any sensors connected to the RS-485 serial bus, called "serial sensors" or "sersens"; plus any custom code that comes up with some number or value based on some application-specific logic, but can then push that value through the I/O subsystem as though it were an actual sensor with limits and Happenings and the like.

There is an entire command set devoted to sending information out through, and receiving information back from, serial sensors (nicknamed “sersens”). [It is appended to this document.](#appsersens)

There are, of course, limited resources on the AssetPack processor, so we can't have a bajillion sensors each recording a tower of sample values. Instead, we have:

* Four fixed sensors: Unit temperature (#0); ANA\_DIO1 (#1); ANA\_DIO2 (#2); and the external power input detection (#3)
* Eight additional sensors with full limit-checking capability, #4-11
* Sensor #4 can be allocated by a System flag to be the tamper switch as a digital input
* Sensor #5 can be allocated by a Mode flag to be the GPS speed, to allow recording and limit checking
* Any sensor >=4 (traditionally #6) can be used for the accelerometer
* Each sensor can record up to 6 historical values
* Each sensor updates its statistics every time the sensor is read, even if it does not record the value as one of its historical values
* An analog sensor (i.e. a sensor that records a numeric value as a reading) is compared to its value limits every time it is read
* An analog sensor is compared to its rate limits every time it is read as part of a periodic (every-X-seconds) sampling
* Any sensor that changes its limit state (green / yellow / red), triggers a corresponding Happening that can induce a response
* The last value read from any sensor, is put into a special variable called last\_sensor\_reading, in case it needs to be read out or manipulated later
* last\_sensor\_reading can also be manually filled with a value (from command ZVV below, or from memory via [Bindump](#bigb)), again so that values from any source can ultimately be treated as a sensor input

See the various "Sensor" commands below for further information on the handling and capabilities of sensors, particular the ‘z’ command immediately below. In these commands, *<sensorhex>* is always a single hex character, 0-F.

z Sensor values (summary)

Note this is a lowercase 'z'.

Show the most recent values read from all sensors. Example:

> **z**

Sensor 0 last @ 134.135 = 996; N = 1

Sensor 1 last @ 161.182 = 545; N = 2

Sensor 2 last @ 167.908 = 543; N = 2

Sensor 3 last @ 0.000 = 0; N = 0

Sensor 4 last @ 0.000 = 0; N = 0

Sensor 5 last @ 0.000 = 0; N = 0

Sensor 6 last @ 0.000 = 0; N = 0

Sensor 7 last @ 0.000 = 0; N = 0

Sensor 8 last @ 0.000 = 0; N = 0

Sensor 9 last @ 0.000 = 0; N = 0

Sensor 10 last @ 0.000 = 0; N = 0

Sensor 11 last @ 0.000 = 0; N = 0

OK1

The AssetPack can record up to 6 readings from up to 12 sensors. (These two numbers can be rebalanced — more readings from fewer sensors — in custom code loads, but cannot be changed over the air.)

* Sensor 0 is always the unit internal temperature
* Sensor 1 is always ANA\_DIO1 on the 6-pin connector
* Sensor 2 is always ANA\_DIO2 on the 6-pin connector
* Sensor 3 is always for external power input detection: when the unit starts and stops receiving charging power over its 6-pin connector, this sensor records a change.
* Sensors 4 and up are for sensors connected via the RS-485 serial port (typically Modbus devices), or for "virtual sensors" used by custom code to record values and statistics.

A 'sensor', as regarded by the AssetPack, is either analog or digital. An analog sensor produces a value, between -32767 and +32767, when read. A digital sensor records the time when it changed state (from 0 to 1, or 1 to 0). A special version of a digital sensor is a Pulse Per Minute (PPM) sensor, where the number of times the input changed state over a period of time is counted, converted to units of PPM, and recorded as though it were an analog sensor with that value[[10]](#footnote-10).

In the example above, Sensor 0 has one recorded value (N = 1), taken at time 134.135, with value 996. Sensors 1 and 2 have two recorded values each, with the most recent ones taken at timestamps 161.182 and 167.908. Sensor 3 shows that no transitions have happened on the external power input line since the readings began. Readings typically begin after the last set of readings have been offloaded from the unit. There is no sensor #4 or higher on this unit, but the values in those sensor records are still read out with this command.

The value of an analog sensor, as noted above, is an arbitrary 16-bit signed value. However, for sensors reading a voltage (particularly Sensor 1 and Sensor 2), that value is calibrated to be in units of milliVolts. For the regular readings of Sensor 0 that the unit makes to monitor its temperature, it records values in units of 0.1C. Digital "values" are 3 for rising (0->1) and 2 for falling (1->0) edges; a value of 0 means no recording took place, and 1 is reserved. For digital PPM sensors, they are treated as analog sensors whose value is in units of PPM.

In the case of serial sensors, values near -32640 are error codes indicating the AssetPack could not complete the communication with the sensor. Most often this is either -32639 or -32647, both indicating a timeout waiting for the sensor to respond. Also note that many serial sensors use the value -32768 to indicate an error on the sensor side.

z*<sensornum>* Sensor values (readings)

Show all available readings from the given *<sensornum>* (in hex). Example:

> **z2**

Sensor 2

167.908 543

146.296 541

0.000 0

0.000 0

0.000 0

0.000 0

OK1

The timestamp, and value, of each recorded sensor value is shown.

z*<sensornum>*s Sensor values (statistics)

Show analog and digital statistics of the given *<sensornum>* (in hex). Note that analog sensors will have nonsensical digital statistics, and digital sensors will have nonsensical analog statistics. Examples:

> **z0s**

Sensor 0 has 72 6 (0): -13

Analog: -23<71, 1041.000, 45455.000

Digital: 2147483647.000 / 0.000, 0 / 0

OK1

Sensor #0 is always the unit temperature sensor, and thus always analog. Most analog sensors are recorded in units of milliVolts, but this one (since it is always a temperature) is recorded in tenths of a degree C.

The 72 is the number of samples that have gone into these statistics. The 6 is an esoteric bitfield showing the current state of the sensor as far as the code knows. The parenthetic (0) is for digital inputs and is irrelevant here. The -13 at the end, on the other hand, is the "value last time" number that will be used for the next rate comparison: this is only checked against, and then repopulated, when a regular sensor gather occurs, as defined by the Mode configuration.

The "Analog:" line, of course, is the relevant one for this sensor. It shows that since the last time statistics were reset (typically the last time they were reported over the air),

* the minimum value seen was -23 (-2.3C), and the maximum value was 71 (7.1C)
* the sum of all samples was 1041 (dividing by the number of samples N reported in the line above, this gives a mean value of 14, or an average temperature of 1.4C)
* the sum of all samples-squared was 45455, which can be converted into a standard-deviation across samples of 20.7, or 2.07C

> **z3s**

Sensor 3 has 0 b (5): 0

Analog: 0<-18504, 0.000, 7.368

Digital: 56908.927 / 7404.087, 2 / 3

OK1

Sensor #3 is always the external power detection input, and thus always digital. The "number of samples used for statistics" 0 and the "last sampled value for rate comparison" 0 are both irrelevant for a digital sensor, and the b continues to be an esoteric bitfield written in hex.

The parenthetic (5), however, now becomes relevant: it is the number of times this input transitioned *triggering the interrupt service routine* in the code. This is important because if the input changes fast, it might change many times before the main code can get around to processing what happened, debouncing it, and so on; but every transition should have triggered the interrupt routine. Thus this number is the most reliable count of actual observed transitions (in either direction) on this line. This number is never reset.

The "Digital:" line is now our focus. It first shows the amount of time, in seconds, that the input was in 0 state / 1 state. This input was usually in the 0 state but was in the 1 state for a little over two hours. Then it shows the number of *debounced transitions* between those two states: first the count from 1 to 0 (called "count0") and then the count from 0 to 1 (called "count1"). The total of these values can be less-than-or-equal-to the number of times the interrupt routine was called since the last report: equal (as in the above example) if the transitions are clean and needed no debouncing, less if the transitions are either fast or noisy.

Important: the count0 / count1 values are reset whenever statistics are reset; the time values are never reset. That is, the count values are "Number of transitions seen since the last report", while the time values are "Total time the input was in this state since the AssetPack was turned on." As noted above, the interrupt count is also never reset, making it "Total number of times this input has transitioned since the AssetPack was turned on."

z*<sensornum>*l[v|r *<limits...>*] Sensor limits

Note the character after *<sensornum>* is a lowercase ell.

Show value and rate limits for the given *<sensornum>* (in hex). If followed by *v* or *r*, change the value or rate limits respectively, in this order, space-delimited:

* Black limit, low: below this value, result is ignored
* Red limit, low: below this value, result is in the red zone
* Yellow limit, low: below this value, result is in the yellow zone
* Yellow limit, high: above this value, result is in the yellow zone
* Red limit, high: above this value, result is in the red zone
* Black limit, high: above this value, result is ignored
* Hysteresis / divisor: a hexadecimal value where
  + the lower 12 bits define the hysteresis of limit checking (i.e. the result must move at least this much further past the limit in the 'good' direction to be considered out of the 'bad' zone)
  + the upper 4 bits define a rolling average behavior: if these four bits are R, then every new sensor value recorded is actually (previous\_value \* R/16) + (just\_read\_value \* (16-R)/16). In other words, these four bits represent the weight (in sixteenths) of the previous value in incorporating the new value into the rolling average. For no rolling average, these bits are zero.
* The green zone is the space between the two yellow limits.
* In case of any ambiguity, black overrides red, red overrides yellow, yellow overrides green.

Any parameter that is not included in this limits list, is left unchanged. Example:

> **z1l** *Note again this is zee-one-ell*

Sensor 1 limits:

Value 0 : 0 : 0 : 0 : 0 : 0 :: 0

Rate 0 : 0 : 0 : 0 : 200 : 10 :: 0

OK1

> **z1lv 0 15 20 80 105 220 8**

Sensor 1 limits:

Value 0: 15 : 20 : 80 : 105 : 220 :: 8

Rate 0 : 0 : 0 : 0 : 200 : 10 :: 0

OK1

> **z1lr -30000 -4.10 -10 10 10 30000**

Sensor 1 limits:

Value 0: 15 : 20 : 80 : 105 : 220 :: 8

Rate -30000 : -10 : -10 : 10 : 10 : 30000 :: 0

OK1

At the beginning, no limits are set for Sensor 1. This means that all values except exactly zero are in the "black zone": ignore value for purposes of limit checking and statistics.

Then, a set of value limits are set: the red zones are 0-15 and 105-220, the yellow zones are 15-20 and 80-105, and the green zone is between 20 and 80. Values below 0 or above 220 are invalid. A hysteresis value of 8 is set, meaning that once the unit perceives it is an a yellow or red zone, it must move at least 8 counts *further* toward green in order to be leave that zone. For example, if the unit reads 25 (green) then 19 (yellow) then 25 again, the unit will still consider that sensor in the yellow zone, because it has not moved past 20+8=28 to get definitively back into the green zone.

Then, a set of rate limits are set: if the sensor changes value more than 10 counts in either direction between two regularly scheduled readings, that is in the red zone. There is no yellow zone, since the red and yellow limits are identical, and red overrides yellow. There is also effectively no black zone, since the black limits are set to impossible values. The hysteresis value is not included in the command, and is left unchanged at zero.

Note that by default, analog sensor limits are set with all Low limits to -32000, and all High limits to +32000. This means that the vast majority of possible values are considered valid and green, while extreme values are likely to be errors of some kind and are ignored.

These limits (and all information regarding sensors) are recorded in RAM for easy tweaking. To persist them to survive a reset, see the [GI](#biggbigi) command.

z*<sensornum>*p[r|d *<params...>*] Sensor parameters

Show the operating parameters of the given sensor *<sensornum>* (in hex). If followed by 'r', change the number of sensor samples to take before one is recorded, and the number of times the sensor must be in a new limit zone before triggering a Happening. If followed by 'd', change the digital debouncing parameters. Parameters are decimal. Examples:

> **z0p**

Sensor 0 70, fl 0, skip 1 of 0, hap@0, deb 0ms, calm 0s

OK1

> **z0pr 4 2**

Sensor 0 70, fl 0, skip 1 of 4, hap@2, deb 0ms, calm 0s

OK1

Sensor 0 is type 0x70, which is [CON\_FUNCTION\_VOLTAGE in hw.h](#appcon), an analog-in type of sensor. (Sensor 0 is always an analog sensor, the unit temperature sensor.) The sensor's "flags", telling the code special variations about how to handle this sensor, is always zero (which it generally is).

The unit can read sensor values, comparing them to limits and adding them to overall statistics, more often than it records a value into its list of 6 to subsequently report. This is expressed by a "skip" value: read and record a value, then skip this many readings (comparing them to limits and adding them to statistics without recording them), then record again. In the first command example above, the skip value is zero (0), and the unit is one (1) reading away from recording a value: skip 1 of 0. This is how a sensor will look when it is primed to start (but hasn't started yet), and skips no readings for recording.

The response to the first command also shows that this sensor will trigger a response (Happening) right away if this sensor goes into its red, yellow, or green limit zone (hap@0). Since Sensor 0 is analog, the two digital values — debounce and calm — are not relevant and will show random data.

In the second command, we set the skip value to 4: the unit will take a reading and record it, then the next four readings will be compared to the limits and added to statistics, but not recorded. In other words, the unit will record every fifth reading it takes of Sensor 0. We also tell the unit that in order to take action (i.e. trigger a Happening) regarding a limit change (going to its red, yellow, or green limit zones), the sensor must have already been in that zone consistently for 2 previous readings: the Happening occurs on the third consecutive reading in that limit zone.

> **z3p**

Sensor 3 51, fl 0, skip 0 of 0, hap@0, deb 2500ms, calm 30s

OK1

> **z3pd 1250 10**

Sensor 3 51, fl 0, skip 0 of 0, hap@0, deb 1250ms, calm 10s

OK1

Sensor 3 is type 0x51, which is [CON\_FUNCTION\_CN (Change Notification)](#appcon), a digital-in type of sensor that detects when the value changes from 0 to 1 or 1 to 0. (Sensor 3 is always a change-notification sensor, the external power input detection.) It also does not skip recording any transitions (it records them all), and is running: skip 0 of 0. Since it is a digital input sensor, the debounce and calm values are important.

* Debounce is the amount of time, after a change happens on the I/O pin, that further changes are ignored because they might just be noise on the line. Once this time expires without any more transitions, the unit decides whether this was a real transition (the input ended up in the opposite state than it started) or was just a noise burst that should be ignored (the input ended up back in the state it started). If it was real, a transition is recorded, and triggers the corresponding Happening. Debounce is measured in milliseconds.
* Calm is the amount of time, after a real transition has been detected, that further changes are ignored because they don't contain more information. If an engine is being started, for example, you only want to be notified of this once, not repeatedly as the starter is cranked multiple times. If, at the end of the calm period, the input ended up back in the original state again — say it went from 0 to 1, was debounced to confirm the transition was real, but then after the calm period it was 0 again — this final change is also considered a single real transition.

In the second command, we reduce the debounce time to 1.25 seconds (1250 milliseconds), and the calm time to 10 seconds.

zr[ *<start\_sensor> <end\_sensor>*] Sensor reset stats

Reset the statistics being tracked on all sensors between *<start\_sensor>* and *<end\_sensor>*, both decimal, inclusive. For analog sensors, this is the sum (for mean), sum-squared (for standard deviation), minimum, and maximum values. For digital sensors, this is the counts of the number of times the sensor has cleanly transitioned to 0 or to 1 (though the total number of transitions in either direction is never reset, always increasing monotonically). For all types of sensors, the count of total sensor readings N is reset.

If *<start\_sensor>* and *<end\_sensor>* are not included, their default values are 0 and 11 respectively, meaning "All of the sensors."

zR[ *<start\_sensor> <end\_sensor>*] Sensor clear limit state

Sets all sensors from number *<start\_sensor>* to number *<end\_sensor>* inclusive, both decimal, to be in the "green" limit zone for both value and rate. In other words, forget everything about whether these sensors were in the yellow or red zone: the next time they are measured, if they are yellow or red, a new Happening signaling this will be triggered.

z*<sensornum>*/[*<connection>*] Sensor reconfigure

Reset the sensor statistics plus the entire knowledge of sensor state (not just limit state, but number of samples taken), and clears the most recently taken value and rate-comparison value. This is generally done to configure a sensor for a new *<connection>* as defined in hw.h.

z*<sensornum>*X Sensor full reset

Sometimes, particularly when a sensor is converted between analog and digital, its information really must be fully reset as though the unit has just started up. This command does that, setting the entire sensor record to zero.

z*<sensornum>*q Sensor quick setup as accelerometer

The accelerometer on the APn board has many good and useful internal features, and with code space on the AP processor at 97% utilization, we have established the architecture that the accelerometer is to be configured using a series of individual SPI push (3) commands, recorded as commands in a SPIF page that is then executed with Run from SPIF (@). That said, it was worth some code space to be able to quickly set up the accelerometer in a generally useful way, as a tilt sensor. This command configures the accelerometer to:

1. Zero out its X, Y, and Z offsets
2. Begin updating at 6.25Hz: fast enough to stay stable, but still only consume 45µA
3. Start measuring

and then sets sensor *<sensornum>* (in hex) to:

1. Be a tilt sensor primed for auto-calibration
2. Zero out any rolling averaging

The first time the accelerometer is read (by a Zg*<sensornum>* command or regular sensor gather), it calibrates as a tilt sensor: whichever axis has the largest magnitude, is considered “down”, and the other two are assigned to be X and Y tilt vectors. These two tilt vectors have their offsets set such that their current values mean “not tilted”, and subsequent reads record these X and Y values in the high and low bytes of the sensor value respectively.

When set up in this fashion, the sensor limit-checking behavior actually operates on the *magnitude(-squared)* of the XY tilt, that is, the number X2+Y2. Also, the rolling average computation (defined above in [Sensor limits](#zlimit)) behaves as expected, independently on X and Y.

z*<sensornum>*m *<numsecs> <delay>* Sensor monitor

This is a convenience function to, every *<delay>* tenths of a second, take a reading from sensor *<sensornum>* (in hex) and display the result. For a tilt accelerometer sensor, it gives the raw two-byte value stored (in hex), the x magnitude, the y magnitude, the net tilt magnitude-squared, and the net tilt azimuth. For all other sensors, it displays last\_sensor\_reading followed by the just-read value — which will generally be identical, unless there is some custom code that manipulates last\_sensor\_reading for this particular sensor.

Z Sensor gather

Note this is an uppercase 'Z'.

Perform a normal sensor-gather step as defined by the currently active Mode. Howsoever the fixed and serial sensors are set to be read in this Mode, including power-on and warmup times, is followed. Results are displayed when available.

Zq Sensor gather (quiet)

Same as the previous commands, without any text output.

Z*<action><target>* Sensor gather (specific)

Perform some *<action>* on a fixed sensor *<target>*. The *<action>* can be:

* s: Set the *<target>* to be an output at 1
* c: Set the *<target>* to be an output at 0
* r: Set the *<target>* to be a digital input and report its value
* a: Set the *<target>* to be an analog input and report its voltage in milliVolts
* A: Set the *<target>* to be an analog input and report its voltage in milliVolts, with ADC calibration

The *<target>* can be:

* 0 (zero): The unit internal temperature sensor
* 1 (one): ANA\_DIO1
* 2 (two): ANA\_DIO2
* 3 (three): The external power input detection line, if read as an input (r); or the sensor power output control line, if set as an output (s or c)
* ! (SHIFT-1): The control of whether ANA\_DIO1 is measuring voltage (0) or current (1)
* @ (SHIFT-2): The control of whether ANA\_DIO2 is measuring voltage (0) or current (1)[[11]](#footnote-11)

Note that the *<action>* can result in nonsensical behavior, such as setting the unit internal temperature sensor to be a digital output. This is a fairly low-level command that is only saved from being a sudo command by being limited to sensors, whose loss is not fatal.

Zg*<sensornum>* Sensor gather (accelerometer)

Read the accelerometer in the way the sensor *<sensornum>* (in hex) is configured. Hopefully this sensor record is configured to be an accelerometer: specifically, that its connection is a CON\_FUNCTION\_ACC\* as defined in hw.h. See [Sensor reconfigure](#zslash) and [Sensor quick setup as accelerometer](#zaccel) above.

ZR*<sensornum>* Sensor gather (record)

When any of the following events occur, the value of last\_sensor\_reading is set:

* a command that [reads[[12]](#footnote-12) a processor pin](#r), analog or digital; its value goes into last\_sensor\_reading
* a [Bindump](#bigb) command; the last 16b word read, goes into last\_sensor\_reading
* a "Sensor gather (set last)" command, see below; an explicit value goes into last\_sensor\_reading
* a [sersens](#appsersens) text value is parsed; this value goes into last\_sensor\_reading
* a Modbus read result is parsed; the last register value received, goes into last\_sensor\_reading

With the ZR command, the value of last\_sensor\_reading is recorded into sensor *<sensornum>* (in hex) as though it were a new reading.

ZVV[x][ *<value>*] Sensor gather (set/show last)

Manually set last\_sensor\_reading to *<value>*. See previous command for an example of how last\_sensor\_reading is used. If the x is present, *<value>* is hexadecimal; if not, *<value>* is decimal.

If *<value>* is not included, the current value of last\_sensor\_reading is shown, along with the current status of the Modbus state machine, which can be:

0: completed successfully

1: awaiting start of response

2: received first byte of response

3: received address and function response bytes

4: received full response short of CRC

-2: response too long

-3: response failed CRC

-4: response came from wrong address or was for wrong function

-9: unit has not recently performed a Modbus exchange

-17: timeout during asynchronous read, awaiting start of response

-18: timeout during asynchronous read, after receiving first byte of response

-19: timeout during asynchronous read, after receiving address and function response bytes

-20: timeout during asynchronous read, after receiving full response short of CRC

-33: timeout during blocking read, awaiting start of response

-34: timeout during blocking read, after receiving first byte of response

-35: timeout during blocking read, after receiving address and function response bytes

-36: timeout during blocking read, after receiving full response short of CRC

ZS*<page>* Sensor gather (sersens)

Run the sersens commands found in SPIF custom page *<page>* (decimal, 0-23). No text output is produced directly; wherever those sersens commands save information (either into sensors to be shown with "Sensor values" commands, or into last\_sensor\_reading to be shown with ZVV), must then be read.

In the odd situation where you're entering this command from the RS-485 port, and it is then sending the sersens commands back out the same port, you will not get a prompt back, as sersens will have taken over the port and put it back to receive mode when it's done. Hit <ENTER> to get the prompt back.

ZModb *<fun> <addr> <reg> [...]* Sensor gather (Modbus immediate)

Immediately perform a Modbus exchange over the RS-485 bus, to perform function *<fun>* on address *<addr>* register *<reg>*[[13]](#footnote-13) (all in decimal). Print out the result received, plus the state of the Modbus state machine, which should be 0 if the command was executed successfully.

The parameters are actually identical to the "SPIF Modbus" commands detailed above. This command is a convenience function that is equivalent to the following:

Gec 1 *Erase SPIF\_CUSTOMS page 1*

Gwc 1/18C1 *Add command to point sersens to RS485 port to SPIF\_CUSTOMS page 1*

GMc 1/*<fun> <addr> <reg> [...]* *Add Modbus command to SPIF\_CUSTOMS page 1*

ZS1 *Execute this command*

*Wait for reply or timeout[[14]](#footnote-14)*

ZVV *Display Modbus state and last\_sensor\_reading*

ZB[c| ]*<binary\_sersens>* Sensor gather (sersens immediate)

Treats *<binary\_sersens>* directly as a sersens command sequence (see Appendix: Serial Sensor Commanding), executing it until there’s a read, then blocking until the read is done or timed out. Sends out a two-byte binary number (little-endian) of last\_sensor\_reading. With optional c, ensures that the command RS485 is set to output before sending last\_sensor\_reading (though note some character, even if just a space, must be present between the ZB and *<binary\_sersens>*).

Zy*<message\_type>* Sensor gather (Send message)

After the next Sensor gather that actually gathers sensor data — Z, ZS, ZModb, or a regularly scheduled sensor gather as defined by the current Mode — is complete, send a message of type *<message\_type>*, in hex. Presumably this message type is set up to send relevant sensor data that were just gathered.

Zd[ *<countdown1> <countdown2>*] Sensor gather (message countdown)

By default, in a given Mode, the lower 16-bit word of its custom[0] value is a countdown of sensor gathers before a Message 1 is sent; and the upper 16-bit word is a countdown before a Message 2 is sent. This is a quick way to implement the concept of “Read sensors every minute to check against their limits, but only report sensor values once a day.” This command displays the current countdown toward zero (send the message) for both of these messages. If the optional *<countdown>* parameters are set, they override the current countdown values.

b*<value>* Buzz

Turn on the buzzer at frequency *<value>*. If *<value>* is zero, the buzzer is turned off. If *<value>* is negative, the [tune corresponding to that number](#appcon) is played.

q*<seconds>* Quiet

The command port is shut down for *<seconds>*, then turns back on. This is useful for turning the command port on remotely: send a q0 command over the air, optionally preceded by a k4 (see next command) to keep the port from turning back off again.

**Command Envelopes**

#*<command><crc>* Parity check

If the first character of a command is #, then the *<command>* portion is run through a CRC check, and compared to the last two *<crc>* characters, which are a hex byte. If they match, *<command>* is executed.

The validation check is

u8 check = 0xFF;

while (cmdlen--) {

check = ((check << 7) | (check >> 1)) ^ (\*cmd++);

}

for all characters in *<command>*, which is all the characters of the received command except the first one (the #) and the last two (the *<crc>*).

This is an envelope that envelops all below envelopes.

]*<startchar>*,*<endchar>*:*<command>* Response limit

When responding to the following command, only include the characters from #*<startchar>* up to but not including *<endchar>*. This can be helpful when the response is longer than the channel conveying that response can handle — particularly, when the command is sent over the air, and the response must find room in the outgoing message queue. If a particular part of that response is what's useful, this envelope can make sure that is the part that gets through.

+[^]*<ms>*:[*<hap\_sched>*] *<command>* Do Later

Perform *<command>*, but *<ms>* milliseconds from now. The unit does this by scheduling a HAP\_SCHED0 event at the indicated time, and then setting the response to that Happening to be *<command>*. The response is assigned to "Mode 0", which means it is executed no matter what mode the unit is actually in (though it does need to be in an active mode for any Happening responses to occur).

The response to this Happening is absolutely set: it replaces any response that may have existed before. To set multiple parallel Do Later commands, include a value from 1 to 14 as *<hap\_sched>*; that changes the event to the corresponding HAP\_SCHED*n* slot.

The optional '^' character tells the unit that the text feedback produced by *<command>*, should be transmitted via the backhaul. This is fairly unusual, but when it's useful, it's really useful.

Note the space preceding *<command>* is critical. After that space is where the parser understands the desired *<command>* starts.

Esoteric: *<hap\_sched>* values beyond 14 are valid, but they start to overwrite Sensor 15’s limit-crossing responses. In the very likely event that there is no Sensor 15, or even if there is, that it’s not performing limit checking, *<hap\_sched>* values up to 24 can be used, and so on for overwriting Sensor 14, etc. See happen.h for details.

@[c]*<address>* Run from SPIF

This command assumes a series of commands, formatted for the OTA incoming message parser — that is, starting with a two-byte big-endian overall length, then each command starting with the command length plus a flag indicating whether or not a response should be queued for transmission back — have been saved starting at *<address>* in the SPI Flash. If the c is present, *<address>* is a decimal number referencing a SPIF custom page; otherwise it is an absolute address in hex. The commands must ultimately end with two bytes of 0xFF, and while they are formatted for parsing by the OTA parser (which can itself run multiple commands in series), each series of commands must be <128 bytes. The overall number of commands that are run by @ is limited only by the amount of SPIF you allocate to the purpose.

This @ instruction then tells the device "Go ahead and run all those commands." The key is that these commands are performed *atomically*, that is, without any other activity in between them. This is helpful for performing a configuration update, where there may be many more commands than can fit in a single OTA message, but you don't want to break the reconfiguration into several messages because the device will have an inconsistent configuration in between them. So instead you break the upload of these reconfiguration commands to SPIF into multiple messages, and then run them all at once. (Often this is done in concert with the [Gc[c] Checksum Check](#biggcc) command, to say "First check that all the commands are in there like I expect, and if so, run them.")

The actual running of the commands from SPIF is done *separately* from the running of this @ instruction, to avoid too much nesting ("Receive a command to run a command to run a command..."). Therefore the return value from this command only means "Okay, I'll run the commands at *<address>* really soon," not that they have already been run.

If the commands loaded in SPIF are time-consuming, which they are likely to be, don't forget to intersperse several [k](#k) commands to clear the watchdog timer.

Appendix: Serial Sensor Commanding

This information is very involved and not currently relevant to the ST9x integration. However, it is potentially useful in future, and is referred to above, so it is included here.

Sensors or other peripherals connected to a serial port — referred to as "serial sensors" or "sersens" in the code — are typically commanded using a sequence of bytes previously written to SPI Flash (by default, at address 0x068000 for commands that are sent every time we are "gathering sensors", and at address 0x069000 for commands to perform a single immediate interaction over the selected serial port). Any byte read from SPI Flash is then pushed out the serial port, with the exception of the below, which are special interpreted commands.

The most common serial port used for sersens, is the RS485 port, shared with the command/response function. Some documentation may imply that "serial port" and "RS485 port" are synonymous, but it's just that RS485 is the most common and default serial port used for sersens sensors.

All values are big-endian raw binary values.

**^X: Connect to channel**

1B value: new value for sersens channel variable

*Set the sersens channel bitfield variable, composed of:*

0x01-0x04 *UART used by sersens: UART1 through UART4*

0x10 *Using ANA\_DIO1 bidirectionally: flip it when sersens changes read/write*

0x20 *Using ANA\_DIO2 bidirectionally: flip it when sersens changes read/write*

0x40 *Using RS485: flip it when sersens changes read/write*

0x80 *Sersens shares channel with command / response (typically UART1)*

*This must be the first command issued, otherwise the sersens system might be connected to nothing. The normal value is 0xC1, which means “Sersens is on UART1, using RS485, and shared with command / response”.*

*This command will always reset the read and write pointers for the target buffer.*

**^A: Set baud rate**

2B value: baud rate in bps

*Set serial port to given baud rate. The special value of 0xFFFE = 65534 means 115,200 baud.*

**ESC (^[): Escape next character**

*Whatever the next character is, send it to the sersens bus, don't interpret it*

**^B: Send binary data**

2B value: Number of bytes to send

followed by that number of bytes

*Send the given number of bytes over the RS-485 bus without interpreting them*

**^M: Send** [**Modbus command**](http://www.modbustools.com/modbus.html)

1B value: Modbus address

1B value: Modbus function[[15]](#footnote-15)

2B value: Register address[[16]](#footnote-16)

2B value: Quantity or Data

If the Modbus function is >=15:

1B value: Number of data bytes

followed by that number of bytes

*Send the given Modbus command, with the CRC auto-calculated. Use ^S to process the response from a Modbus read command.*

**^S: Monitor for Modbus response**

1B value: [Starting] sensor # to record the response(s) to

1B value: How to parse the incoming register data

*The parsing parameter has 7 fixed valid values:*

*0: Parse each 16-bit Modbus register as a normal 16-bit register*

*2: Parse two consecutive Modbus registers as a 32-bit Big-endian signed int (MSW[[17]](#footnote-17) first)*

*3: Parse two consecutive Modbus registers as a 32-bit little-endian signed int (LSW first)*

*4: Parse two consecutive Modbus registers as a 32-bit Big-endian unsigned int (MSW first)*

*5: Parse two consecutive Modbus registers as a 32-bit little-endian unsigned int (LSW first)*

*8: Parse two consecutive Modbus registers as a Big-endian single-precision float (MSW first)*

*9: Parse two consecutive Modbus registers as a little-endian single-precision float (LSW first)*

If the parsing parameter is between 2 and 7:

4B value: unsigned divisor to scale 32-bit value to 16-bit signed AssetPack sensor value

4B value: unsigned subtrahend to subtract from the scaled value

If the parsing parameter is between 8 and 15:

4B value: floating-point divisor to scale floating-point value to 16-bit signed AssetPack sensor

4B value: floating-point subtrahend to subtract from the scaled value

The divisor and subtrahend are encoded little-endian. (Sorry.)

*Listen for the response to a command sent with ^M. What comes back is a series of 16b register values, which is great because the AssetPack’s sensor system itself is based on 16b signed sensor readings. If the parsing parameter is 0, the incoming registers are fed into the given sensor #s, sequentially. (If the sensor # is higher than the number of sensors the AssetPack can handle, the value is read but not recorded.)*

*The trick comes when a manufacturer uses the 16b Modbus registers to hold 32b values. We then need a way to identify these 32b values, and from there to scale them into the 16b sensor readings the AssetPack works with.*

*Two important notes:*

1. *This scaling is only required when the AssetPack itself needs to understand the sensor readings, primarily to run limit checks on them, or calculate statistics on them. If the AssetPack is simply recording and relaying the Modbus data, it is fine to read the two Modbus registers into two consecutive sensor #s, send both sensor values OTA, and let the server take care of the parsing.*
2. *If you want both range-checking and the full results from the serial sensor, use two Modbus commands to populate three sensor #s: one command populating two sensor #s for the full data (parsing parameter 0, ‘just record it’), and one command populating one sensor # with scaling for comparison against limits.*

*The scaling works the same way no matter the parsing method; the difference is just in the format of the numbers involved (fixed-point vs. floating-point, signed vs. unsigned). The scaling is:*

*reading = (32b\_read\_from\_sensor / divisor) - subtrahend*

*It is a linear mapping of the value the serial sensor is actually supplying, onto the 16b signed range the AssetPack uses for limits and statistics. The important thing to keep in mind, bluntly, is that any physical sensor in an outdoor environment is only going to have 16 bits’ worth of actual data to supply; everything else is noise. (Is this temperature sensor really giving accurate values down to 0.0000001 degrees?) This scaling is meant to pull out that actual range of 65,000 distinct values that are actually meaningful.*

**^L: Flip to reading**

*Set the RS-485 driver (or other half-duplex line) to receive mode instead of transmit. Usually followed by a ^P command to wait for some time, and then read what was sent. Note: use the ^R command instead to make sure the reading starts from this moment.*

**^R: Reset read pointer and flip to reading**

*Same as ^L, but also ignores any information / noise which may have come along the RS-485 bus before this moment*

*Note: ^S, ^L, and ^R all flip a half-duplex connection (viz. the RS-485 bus) to receive. Any subsequent command to send a byte, flips it back to transmit. The declaration of what I/O should get flipped, if any, is done via the sersens channel variable — see ^X command above.*

**^P: Pause**

2B value: number of milliseconds to pause for

*The interpreter stops reading commands for this many ms, then resumes*

**^Z: Parse text response**

2B value: number of milliseconds to continue searching for

1B value: flags

NUL-terminated text format string

*Starting at the current read pointer, apply the scanf-style format string, looking for numbers to parse as sensor readings. This is going to take a while to walk through, so strap in.*

*The “number of ms to continue searching for” lets the system keep applying your format string to the text coming from the serial sensor, over and over as new text comes in, so that a result is found as soon as possible — but no longer than this timeout. The other way to do this is to issue a “Pause” command, then following it with a “Parse text response” with a timeout of zero. The actual sensor involved will likely favor one or the other of these approaches: look repeatedly until the parser catches the data, or, wait until you know the data are in the buffer and then parse it once.*

*There are currently two flags:*

*0x80 Attempt to parse with every character that arrives over the serial port (1) or only when a newline character arrives (0). A newline is either \r or \n, so for sensors that send both, two parsings will be attempted in succession. A parsing is also always attempted at the timeout, irrespective of what has come in over the serial port.*

*0x40 If not all the values could be parsed by the timeout, keep the ones that were found (1) or abandon them all and consider this a non-reading (0).*

*Finally we get to the meat of the command, which is the format string letting the AssetPack know how to find the actual sensor readings. Up to 8 numbers can be read using a parsing format string. All numbers are parsed as floating point, are then scaled, and are then recorded as 16 bit signed values (-32767 to +32767) into a given sensor #.*

*The important keys in the parsing format string are:*

%f[#,divisor,subtrahend] *Parse a floating-point number at this point, divide it by* divisor*, subtract* subtrahend*, and enter it as a reading for sensor* #*.* divisor *and* subtrahend *are both floating-point, and they must map the possible values sent by the serial sensor to the [-32767, 32767] range of an AssetPack sensor. Values outside these bounds will be saturated, i.e. if after the parsing the value is negative three million, the reading put in the sensor will be -32767. Default divisor is 1, default subtrahend is 0.*

? *Metacharacter meaning “Any character other than a newline”*

# *Metacharacter meaning “Any digit 0-9”*

\* *Metacharacter meaning “Any number of repetitions of the previous character”*

| *Metacharacter meaning “Parsing break: allow any number of characters between what’s on the left, and what’s on the right”*

\ *Metacharacter meaning “Treat the following character literally, not as a metacharacter”*

@ *Metacharacter meaning “Parse the actual number signaled by %f, starting here”. This allows the parser to actually inspect the format of the number being parsed, before going back and parsing it.*

*Let’s say the string in the serial input buffer is*

abbacy 14 abc 6.28\r\n\0

*… and it’s the 6.28 we want to read. A format string of “*abc %f[4,0.01,300]*” would find that contiguous string abc<space>, but then try to make sense of a number containing a lot of extra mess:*

abbacy 14 abc 6.28\r\n\0 -> 6.28\r\n

*Adding a terminating delimiter, “*abc %f[4,0.01,300]\r*”, gets us where we need to be:*

abbacy 14 abc 6.28\r\n\0 -> 6.28 ✔

*This value of 6.28 would be divided by 0.01, making 628; then subtracted by 300, making 328. This value 328 would be inserted into Sensor 4 as a new reading.*

*We had a customer give us a sample of their serial sensor’s output; here are some excerpted lines:*

340318 28912 2018/11/01T19:43:06z:A: PM1.0: 3 2.34 (2.34) PM2.5: 3 3.19 (3.19) PM10: 4 3.21 (3.21) T: 0 0.00 H: 0 0.00

340329 28912 2018/11/01T19:43:06z:B: PM1.0: 3 3.17 (3.17) PM2.5: 4 4.70 (4.70) PM10: 4 5.50 (5.50) T: 0 0.00 H: 0 0.00

341143 30024 2018/11/01T19:43:06z:A: 11230 3

341986 30024 2018/11/01T19:43:07z:A: 842 3

342898 30024 2018/11/01T19:43:08z:A: 913 4

350672 30000 2018/11/01T19:43:16z:B: 11193 4

351581 30024 2018/11/01T19:43:17z:B: 909 4

352420 30024 2018/11/01T19:43:18z:B: 839 4

*We don’t know what in here is the meaningful data, but we can show some format strings for various possibilities. Let’s say “A” should go into sensor 4, and “B” into sensor 6.*

*Is the meaningful value, the second value after PM1.0? Try*

*“*A: PM1.0: # %f[4,0.01,0] |B: PM1.0: # %f[6,0.01,0] *“*

*Note the trailing spaces after the two %f’s are critical, because they bound the far end of the number (“the number ends when the space shows up”). If it’s always a number with four characters, we could alternately do*

*“*A: PM1.0: # %4f[4,0.01,0]B: PM1.0: # %4f[6,0.01,0]*“*

*If, on the other hand, the PM1.0 values are not what we want, but rather the numbers on lines on their own, we could parse*

*“*A: @#\* #\r%f[4,1,0] |A: @#\* #\r%f[4,1,0] |A: @#\* #\r%f[4,1,0] *“*

*“*B: @#\* #\r%f[6,1,0] |B: @#\* #\r%f[6,1,0] |B: @#\* #\r%f[6,1,0] *“*

*This reads three readings into Sensor 4 in turn, followed by three readings into Sensor 6. Note the use of the ‘@’, meaning “match this pattern of A<colon>, a space, an unknown number of digits, a space, a single digit, and a carriage return. If you match it, the number for that %f following is back here at that multi-digit number.”*

*If those 5-digit numbers aren’t what we want, we just want the first 3-digit value in each sequence, we could have*

*“*A: @### #\r%f[4,1,0] |B: @### #\r%f[6,1,0] *“*

*The critical functions from the firmware have been excerpted to a separate program called sersensParse.c, which can be compiled on a computer and used to test candidate parsing format strings against candidate serial sensor output files.*

**0xFF: Done and shut down**

*Release the I/O pins claimed by the ^X command and stop parsing sersens commands. This need not be an explicit command: by nature, when the last serial-sensor character is written to SPI Flash, it will be followed by the empty byte 0xFF, ending the command sequence.*

**0xFE: Done**

*Same as previous command, but does not release the I/O pins. Useful if the external sensor is powered up or active even in between sersens interacting with it.*

1. Default / example message type meanings are shown for clarity [↑](#footnote-ref-1)
2. This is a separate message queue in the SPI Flash where bursts of high-value data can be stored, and then explicitly read out by the DeviceManager server. It is semi-custom to applications which have this kind of burst data and not normally used. [↑](#footnote-ref-2)
3. A hexadecimal value (0-9,a-f) may also be used [↑](#footnote-ref-3)
4. For the Iridium 9602 modem, exchange codes <=4 denote success, higher values denote failure. [↑](#footnote-ref-4)
5. Different modems report RSSI differently; Iridium uses a “bars” scale of 0-5, like a cell phone screen [↑](#footnote-ref-5)
6. This is the actual register number value to be sent over the wire. Awkwardly, this is often 40001 *less* than the number shown in documentation. That is, if your sensor documentation says a register is at location 40030, the number actually sent over the wire is 29 (0x1D) — the 40000 just means "holding registers", which are 1-based, while the value on the wire is just a 0-based register number. [↑](#footnote-ref-6)
7. strictly speaking, greater than or equal to NUM\_SENSORS, which is usually 12 but cannot be higher than 16 [↑](#footnote-ref-7)
8. This is the actual register number value to be sent over the wire, see top footnote above. [↑](#footnote-ref-8)
9. If *<func>* has its bit 0x40 set — that is, a function number of 0x45 (69) instead of 5 — the last\_sensor\_reading is used instead of *<value>* when the command is sent over Modbus [↑](#footnote-ref-9)
10. Note that one reading slot is reserved for keeping track of these calculations, so PPM sensors can hold one fewer reading than other sensors — 5 readings instead of 6. [↑](#footnote-ref-10)
11. Note: for stock AP3 boards, only ANA\_DIO2 can be set to measure current [↑](#footnote-ref-11)
12. When a processor pin is set, it is also then read, so this includes commands that set a pin [↑](#footnote-ref-12)
13. This is the actual register number value to be sent over the wire. Awkwardly, this is often 40001 *less* than the number shown in documentation. That is, if your sensor documentation says a register is at location 40030, the number actually sent over the wire is 29 (0x1D) — the 40000 just means "holding registers", which are 1-based, while the value on the wire is just a 0-based register number. [↑](#footnote-ref-13)
14. This is actually different than the ZS1 behavior, which is asynchronous [↑](#footnote-ref-14)
15. If the bit 0x40 is set in the function byte, it is a flag to jam the last\_sensor\_reading variable into the Write Data parameter of the Modbus command. This provides a clumsy but viable way to send non-constant data from some source into a Modbus device. [↑](#footnote-ref-15)
16. This is the actual register number value to be sent over the wire. Awkwardly, this is often 40001 *less* than the number shown in documentation. That is, if your sensor documentation says a register is at location 40030, the number actually sent over the wire is 29 (0x1D) — the 40000 just means "holding registers", which are 1-based, while the value on the wire is just a 0-based register number. [↑](#footnote-ref-16)
17. Modbus itself is Big-endian, so within each 16b word, the bytes are sensibly ordered from most to least significant. But in a desperate play to regain stupidity, some vendors then take those *words*, and pop them out least-significant-first. Anthony calls this “Middle-endian”, a good name. (still stupid) (But it’s). [↑](#footnote-ref-17)