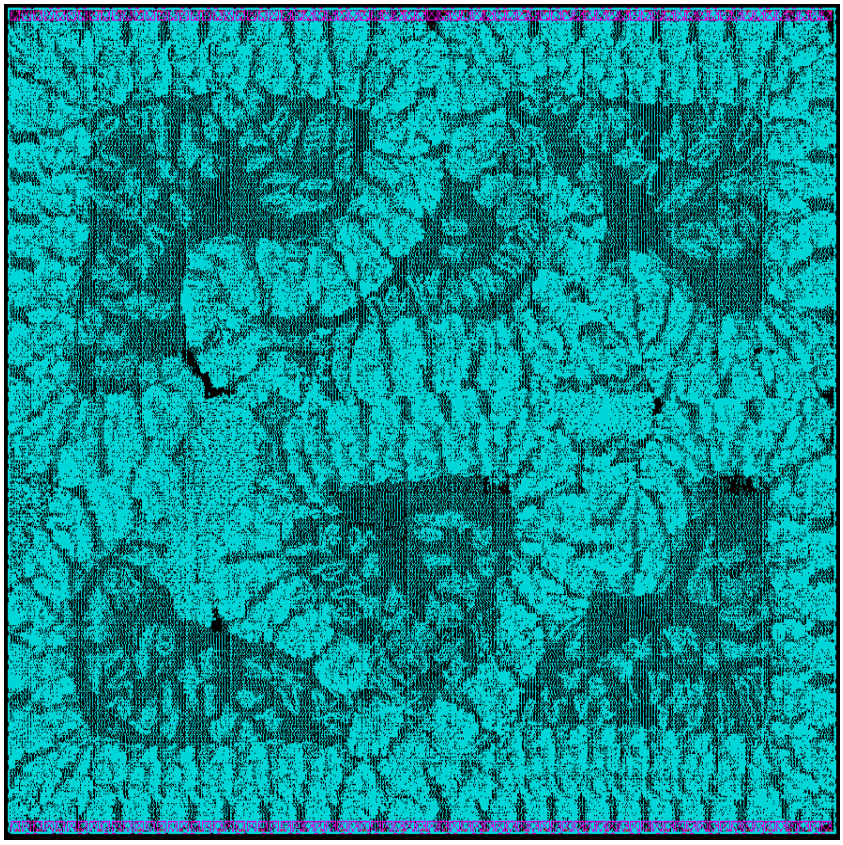
Golden Nonce (GN) ASIC

Interface Protocol Guide



HashFast

Revision History:

Date	Revision	Comments
May 21, 2013	0.1	Initial release
Oct 21, 2013	0.2	Initial public release

On the cover:

An early placement map of cells in an individual HashFast search core, circa July 2013

Table of Contents

1 How to use this document

The user guide describes the Interface Protocols used at various levels for the HashFast Golden Nonce (GN) ASIC, and for board assemblies and related products produced by HashFast. The intended audience is:

- Driver developers, who wish to write device drivers for mining applications
- Third party OEM's, who are deploying HashFast ASICs or IP in their own hardware, and must write interface firmware or drivers associated with their value added products
- Technical users who wish to know more about the mining products they have purchased
- Any user who simply wishes to know more about HashFast products

This is an interface protocol document, it does not provide electrical design information associated with the HashFast GN ASIC.

If you are:

A Driver Developer

You will be primarily interested in Chapter 3, which describes the two different USB protocols that may be used to interface with HashFast products. You will almost certainly elect to use the Global Work Queue (GWQ) protocol, which operates at a high level of abstraction, does not require the driver to control individual cores, and makes low demands on host resources by deriving work internally from a relatively few number of host driver provided work items. If you prefer, you may instead elect to use the USB Mapped Serial protocol (UMS), in which case you will be writing more driver code in order to keep work queued to each individual search core across all ASIC's.

You may occasionally refer to Chapter 2 in order to better understand the low level operations that are occurring, or for those cases where the USB protocols refer to the underlying ASIC communications that are used to bring about actions.

You will definitely be interested in the header file listed in Appendix C at the end of this document, because this header file is available open source (GPLv3) from HashFast and contains useful structure definitions that will help you write your driver.

An OEM developing boards that use HashFast GN ASICs or IP

Unless you have licensed and intend to use firmware from the HashFast reference design, you will primarily be interested in Chapter 2, which describes in detail the low level serial protocol used to interface directly with HashFast GN ASICs. This Chapter includes information about all aspects of the low level communications, including peripheral interfaces such as general purpose I/O capabilities that are available in the HashFast GN device but not necessarily used in HashFast modules. You will typically use an embedded CPU of some form to interface with the GN ASIC chain at this low level.

All users

You may wish to start with Chapter 4, which provides an overview of the GN ASIC parameters and the board assembly used within various HashFast products.

2 ASIC Serial communications protocol

> *Note: If you are a user or a driver developer for HashFast products, you will be using one of the USB interface protocols, as described in Chapter 3. Use the material in this Chapter only as reference material about the underlying hardware operations.*

The HashFast GN ASIC supports a communications protocol for daisy-chaining multiple devices together, and communicating with a host system or board master. The physical ASIC interface may be either serial (UART, 1.8V levels) or parallel (byte wide). Most implementations will use the serial operating mode. The serial interface may be used with any number of different physical interfaces to yield different implementations, for example:

1. ASIC to ASIC interfaces consist of direct wired connections between devices
2. Direct connection to a board level micro-controller, or host connected through a USB capable micro-controller. In this case the micro-controller could actually change the interface protocol into something else. This is the method used in HashFast mining products.
3. A USB to serial converter, such as an FTDI FT232R, may be directly connected to the ASIC, to allow communication with a host through a standard USB connection. This means to achieve USB communications is an expedient but highly restrictive option, and is not recommended for new product development.

This Chapter deals with the ASIC serial protocol only, the low level protocol that operates directly between inter-connected HashFast GN ASICs. The parallel interface method is more usually employed for testing, and will not be discussed further in this document. All command and response frames may be employed with either interface, although the ability to change baud rates is not relevant when using the parallel interface connection method.

2.1 Topology

Figure 2.1 illustrates how a “system” of one or more HashFast ASICs may be assembled to form chained systems of very high hash capability that are accessible from a single, simple interface.

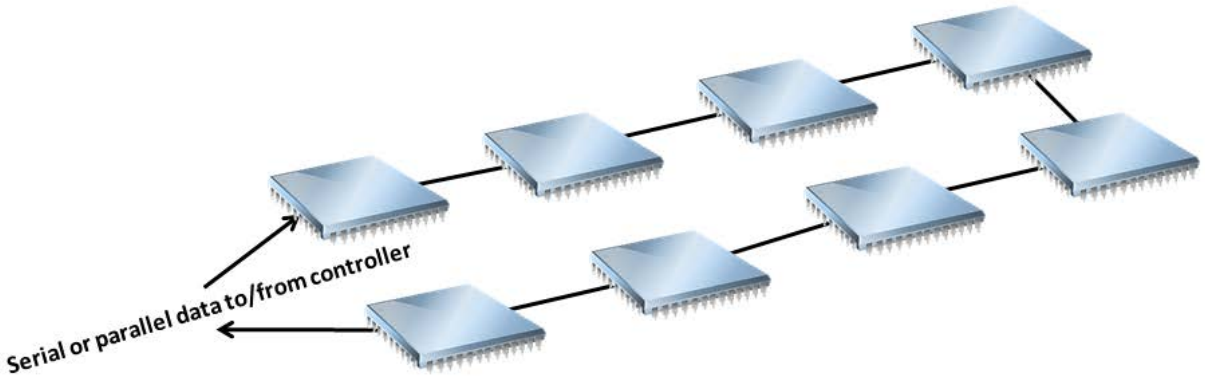


Figure 2.1: Interconnection of multiple GN ASICs

2.1.1 Serial interface characteristics

The serial interface ports consist of a single transmit data pin (txd) and a single receive data pin (rxd). There are no flow control or modem control signals. Standard baud rates between 9600 baud and 3 Mbaud are available, or any non-standard rate in between. The host system may issue a command which re-programs baud rate. The default baud rate on power-up may be configured between one of eight rates, based on the power-up state of three configuration pins (gpi[7:0]).

Serial transmissions consists of

- One start bit
- Eight data bits
- One stop bit

and data is transmitted LS bit first. There is no parity support.

A logic level of “high” denotes an inactive state on the serial wire. The “start” bit is sensed when a high to low transition occurs on an otherwise idle port. Following the start bit, the 8 successive data bits are each sampled in the middle of the bit time slot, and with a logic “low” value corresponding to a bit value of 0, and a 'logic “high” value corresponding to bit value of 1.

2.1.2 Protocol frame header

Messages on the serial link are performed through the use of protocol frames, which consist of a fixed length header which is always present, optionally followed by a variable length data frame, which is often not present. The header consists of a fixed code preamble, six bytes of content, and a CRC-8 trailer which is used to detect transmission errors in the header information.

Message frames pass through the serial chain and may, depending on the message type, either be removed by the destination ASIC, or passed all the way through the chain and wind up being received back by the host or board level micro-controller.

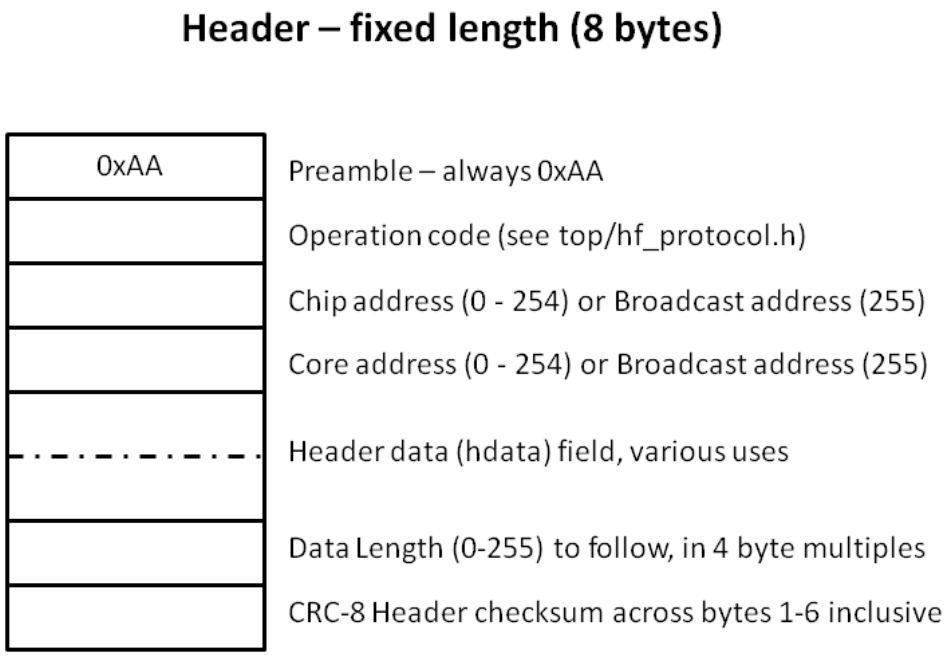


Figure 2.2: Header format

Figure 2.2 shows the format for the fixed length, 8 byte header frame. The fields are as follows:

- A fixed preamble code, which has the hexadecimal value “0xAA”. When idle, the receive framer looks for bytes which have this value. When a byte of this value is received, a timer is started and the receive framer examines the subsequent seven bytes to see if they constitute a valid protocol header. If seven more bytes are not received before the timer expires, then any received data is discarded and the framer once again looks for a preamble. The timeout value is programmable, in terms of character times, but is generally set to a relatively low value. Default timeout is 32 character times.

- An “operation code” byte, which designates what the message means. Operation codes are detailed in a later section.
- A “chip address” byte, used to address one of up to 255 ASICs in a single chain, through addresses 0 – 254 inclusive. Address 255 is the “broadcast” address, used to direct actions at all devices in the chain.
- A “core address” byte, used to address one of up to 255 hash cores within any given ASIC. Core address 255 is reserved as an internal “broadcast” address, to direct actions at all hash cores concurrently.
- Two bytes for header included “additional information”. The meaning of this field, nominally called “hdata”, is operation code specific.
- One byte indicating the length of the data frame, if any, to follow, in units of 4 byte chunks. If this value is zero, then no data frame is to follow. A data frame consisting of up to 1020 bytes may be specified by lengths within this field up to 255 inclusive. Data frames are postfixed by a CRC-32 trailer, making the maximum length of a data frame 1024 bytes. Data frames are described in a later section.
- A final byte, consisting a CRC-8-CCITT check field, computed across all preceding bytes except the preamble, using the generation polymomial $x^8+x^2+x^1+1$. This gives a good degree of protection against undetected errors across the header fields.

There are three inter-related protection mechanisms to ensure that noise and transmission errors cannot compromise the integrity of the system:

1. Each device error checks the preamble value and the header CRC, and if an error is found, the header is discarded.
2. A timeout is also employed when a header is being searched for, so that partial headers (due to noise) are discarded.
3. Finally, critical system information held within each ASIC, such as the ASIC's address, can only be modified when the ASIC is in a “privileged” access mode. Specific header operations are required to place a device into “privileged” mode, and devices will normally be taken out of “privileged” mode and placed back into “user” mode for normal operation of the chain, i.e. hash computational work.

   As such, in the highly improbable case of an erroneous header generated by noise being accepted by the framer, if the device is in user mode, no actions that could compromise the integrity of the system would be accepted.

Note that the typical extent of the serial interface is almost always confined within a chassis or board assembly, so transmission noise can be will controlled and high speed rates may be comfortably achieved.

2.1.3 Flow control – there is none

The host system manages workload across the hash engines. Each hash core may be performing one active search, and have one search pending – which will automatically commence after the conclusion of the current hash job. Through mechanisms described later, the host will always know when an available "slot" exists to queue work to a hash core. As such, the host *never sends work which cannot be accepted.*

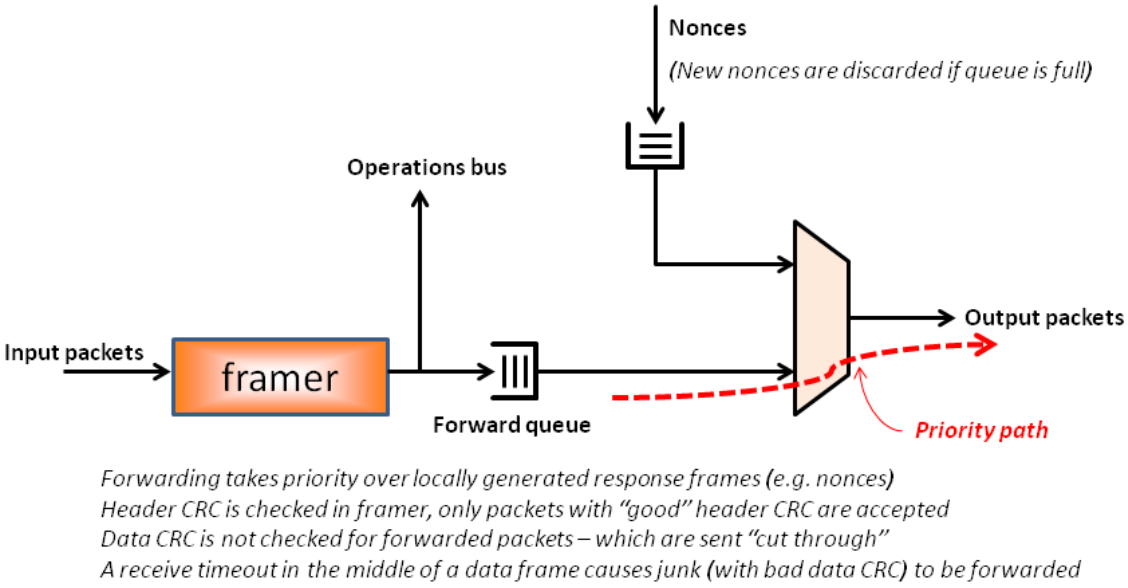


Figure 2.3: Transmission path through an individual GN ASIC

Figure 2.3 illustrates the transmission path taken by frames as they enter each ASIC. Incoming headers are detected by the framer, and once accepted as valid, may perform actions on this device (if addressed to this device, or the broadcast address) and/or be forwarded through an output multiplexing stage to the next device in the chain, or be discarded (if addressed to this device and be of an operation type which does not lead to an immediate response).

The "Forward queue" in this figure denotes a queue which is used to store packet words in the event the packet has arrived while an output frame is being generated by this device, generally from the indicated top queue containing a stream of candidate Nonces. Output frames are infrequent and short, and the size of the forwarding queue is based on the maximum length of an output frame that can be independently generated by any device.

Whenever there is a packet pending in the forward queue, the output multiplexer will complete any frame which is being internally generated, and immediately switch to the forward queue in order to transmit packets which are held for forwarding. This forwarding process will continue until (a) the forward queue is empty, and (b) a complete packet, including any packet data, has been sent, i.e. we have not starved the forward queue mid packet.

If a packet is truncated in the middle of the packet data fragment for some reason, this system could

starve and back up while the forwarding mechanism is waiting for the remainder of a data packet that will not arrive. For this reason, a timeout mechanism is wrapped around the data portion of any ingress packet which is being forwarded, and if a timeout occurs, the framer will forward substitute data along with a CRC-32 check which will always result in failure, causing the data to be discarded further down the chain.

Apart from the headers, packets are forwarded on a “cut through” basis, i.e.

- The header is stored and checked for integrity before a forwarding operation starts, so the 8 byte header does not “cut through” into the forwarding queue before it is checked, but
- The packet data is passed on without being first fully checked and stored, so the data portion of a packet is forwarded “cut through” with minimum latency.

What happens if the nonce queue fills up? Well, in that case subsequent nonces get dropped, and a statistics counter keeps track of such events. If this happens, however, it is indicative of a mis-tuned system. Consider the following case:

- The ASIC contains 8 cores
- Each core can sweep through the full 2^32 nonce range in 2 seconds
- There are 8 ASICs in the chain.
- Difficulty is set such that on average only two nonces are turned up for every 2^32 sweep. (This might be a reasonable number for a mining pool application, where difficulty is artificially eased for workers).

That's 64 cores, total, producing on average 128 nonces every 2 seconds. Although nonces can be batched for sending back to the host, let's assume they all go back in separate messages to the host. Each nonce message will be 20 bytes long (see later), so 128 * 20 / 2 = 1,280 bytes per second of egress traffic on the chain. For a serial interface operating at, say, 1 Mbaud, at 10 bits per byte that's 12.8 msec of egress activity every second, or 1.28% of the link bandwidth. The incoming messages to set up these 64 jobs are inconsequential – since there is only one message every 2 seconds to each core. Monitoring and statistics traffic is also inconsequential.

Each core has a small amount of egress buffering, and these buffers are emptied round robin into a single egress buffer that is illustrated in Figure 2.3 (marked “Nonces”). The probability of these “Nonce” buffers filling up, for appropriate ASIC difficulty levels, is near enough to zero. If a user is losing nonces, it is indicative of a problem in the application or mining pool client setup, in that the difficulty is set too low.

2.2 Core operations within the ASIC

Each hash core inside the ASIC has a “core address”. This address defaults on power-up, with cores being numbered sequentially starting at zero. The maximum number of cores allowed is 255, although practical devices today have fewer than this. The GN ASIC has 96 cores. A common interface, driven by the framer, can push data into an input staging area of each core, but only the “addressed” core will accept the incoming data. A core “broadcast address” of 255 is allowed, in which case the same work

can be pushed into all cores on the device, but this is not used in normal operations, because it is hardly useful to solve the same problem more than once! Group methods are available (described later) to allow setting up multicast groups that modify the work (by splitting nonce ranges and/or ntime rolling). This is the primary application of broadcasting.

Figure 2.4 illustrates the arrangement of how work gets into and out of a single core. A work assignment for the core will take place in the form of an OP_HASH operation (see Section 2.4.2.1), which loads a midstate, block data, nonce range, difficulty target etc. into the core. This data is shifted in, one byte at a time, at the serial link byte rate, into a holding register. Only when all data is clocked in, and the CRC-32 check verified by the framer, is the core told to “go”, shift the work in and commence searching.

The hash core operates at a much higher clock rate than the framer, queuing and de-queueing logic, and results (Golden Nonces!) from the hash core are first buffered through a few stages operating at the hash core rate, before traversing the clock boundary again and entering the core's egress queue, which operates at the lower core clock rate (typically 125 Mhz).

Each hash core has room for one pending job, in addition to the search operation it is currently working on. As soon as a search operation completes, if there is a pending job waiting in the hash core's input queue, then that job is loaded and a new search commences. Mechanisms are provided to allow the host to know when hash core input queues become empty, so that the host can load up new jobs at the relatively sedate rates of the serial link, without having any hash core sit idle. In this way, hash cores can be kept working continuously, fulfilling their purpose!

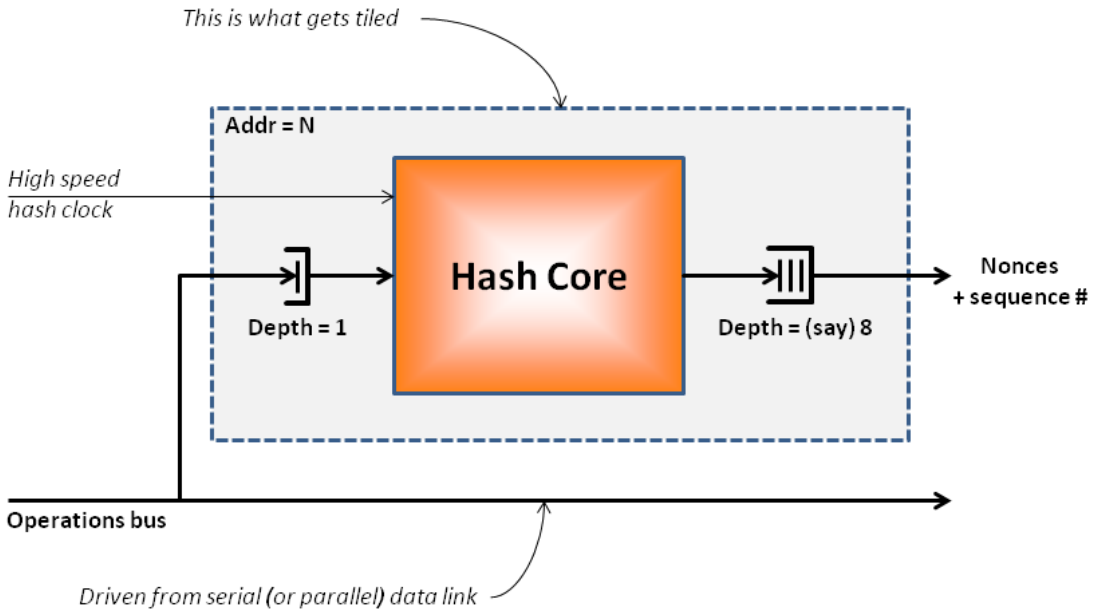


Figure 2.4: Operations with an individual hash core

An array of hash cores is managed by addressable input, and multiplexed output. This works due to the relative infrequency of output. Consider Figure 2.5, which illustrates an array of tiles, with a common input side operations bus.[1]

The operations bus has an “address”, and “data” component, together with a “data_valid” signal which pushes the work data into an addressed core, one byte at a time, as the input framer receives the data. Only the addressed core actually accepts the input data.[2]

When all input data is shifted in, the framer validates the CRC-32 check at the end of the data packet, and only then the work is handed off to the core.

On the output side, each core has an output nonce “queue”, and after down-shifting to the lower clock speed domain, outputs are multiplexed into a single result core for the ASIC. This multiplexing operation is illustrated in Figure 2.6 on page 15.

2.2.1 Cores may be readdressed

Note: The capability to re-address cores is deprecated in the GN ASIC.

By default, all cores assume sequential addresses on power up. There are some circumstances where it may be desirable to re-address cores. These circumstances include:

- System cooling is such that not all cores can be operated continuously. A device in a liquid cooling situation may be operable with all cores active, but in an air cooled situation may need to be de-rated. A board level controller may perform the re-configuration necessary to limit the number of cores visible to the user, to prevent “abuse” of the product by the user.
- Parts sorted with one or two cores not operating properly may still be recovered as useful product, in which case cores which are inoperable can be re-addressed outside of the operating address range, and downstream cores re-addressed to form a product with less cores, but with otherwise identical characteristics.

1 This is actually NOT how the GN ASIC is organized internally, but it is adequate for illustration purposes.
2 Or all cores, it the broadcast address is used.

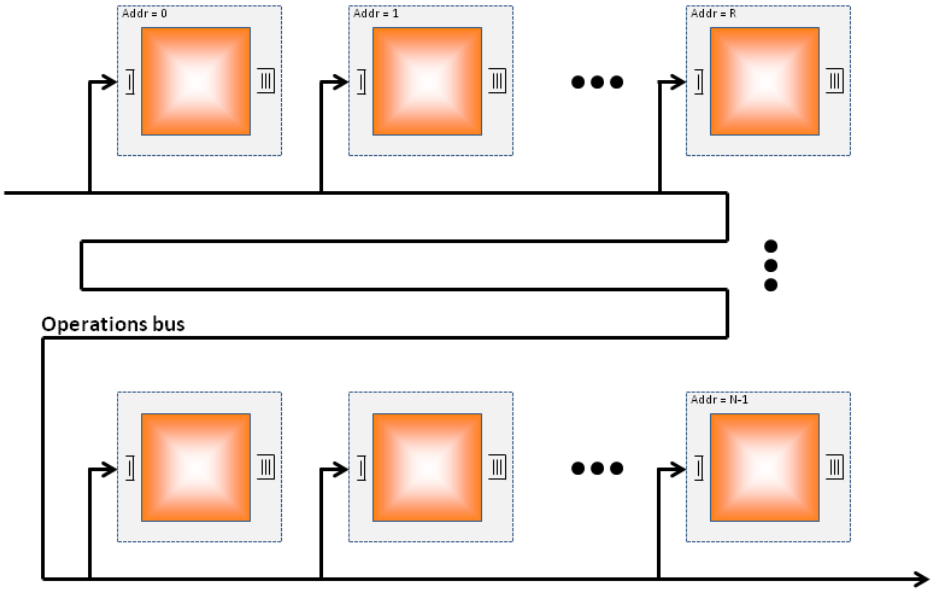


Figure 2.5: Operations bus is common, and queues work to cores by address

2.3 Packets with data

Some packet types have data. The fixed 8 byte header is the same, but it is followed by a data block, which is always a multiple of 4 bytes in length, and is finished with a CRC-32 integrity trailer.

The presence or absence of data is indicated by the field at byte offset 6 in the header. A zero value in this field, which is validated within the header CRC, indicates that no data follows. A non zero value for this field indicates the presence of data.

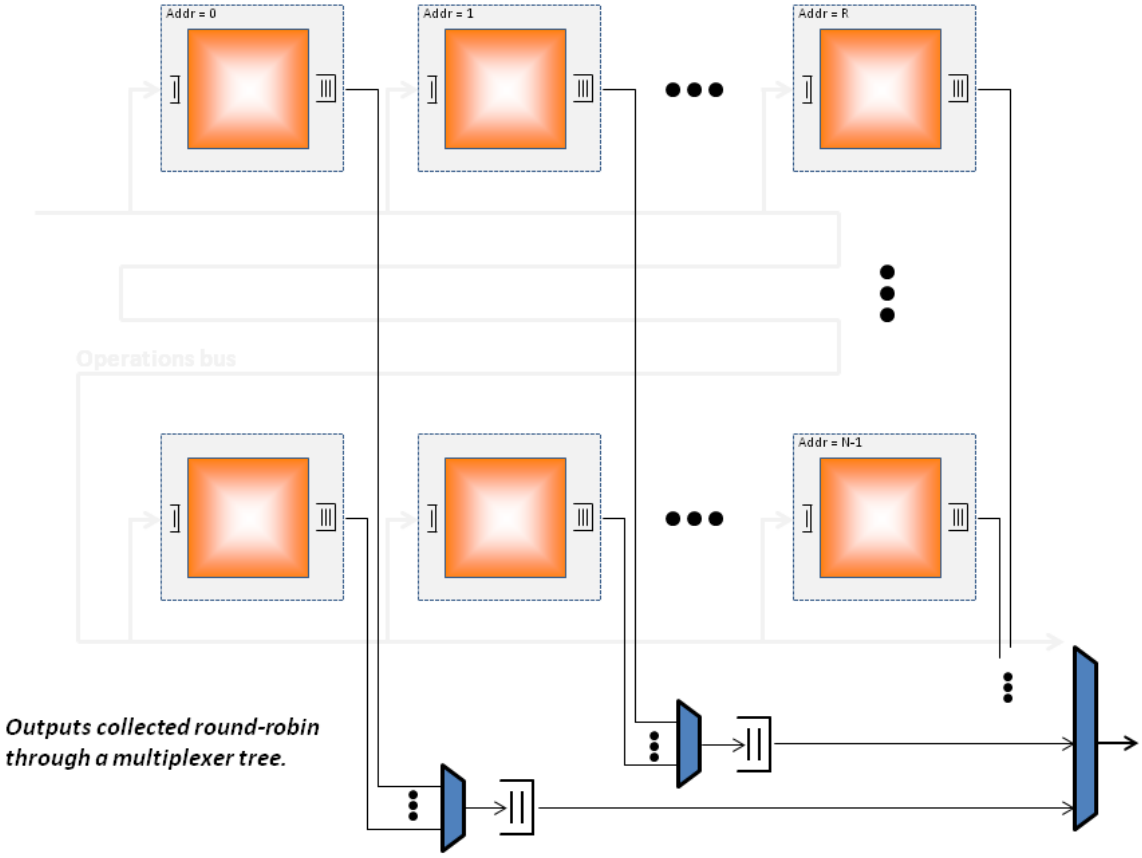


Figure 2.6: Hash core output multiplexing

2.4 Operations

Packet level operations are based on the operation code contained in the header. Only verified headers are considered, headers which do not pass the CRC-8 validation check are discarded.

Any valid headers which contain operation codes that are not part of an ASIC's command set are forwarded without modification. This enables potential mixture of existing and future generations of ASIC in the same chain – an unlikely but nonetheless plausible event.

Apart from some isolated cases (e.g. an address cycle), the host can consider itself to be communicating with cores and devices in an arbitrary network manner, where it has point to point communications capability with any device in the network, and broadcast capabilities across devices. From a transaction standpoint, the "daisy chained" nature of the device inter-connectivity is largely immaterial.

The following sections describe the supported operation codes for the GN ASIC, and illustrate the transactions as "command-response" or "unsolicited response" interactions. Fields in the packet header will be referred to by name, in the context of the following "C" data structure:

```
struct hf_header {
        uint8_t  preamble;                      // Always 0xaa
        uint8_t  operation_code;
        uint8_t  chip_address;
        uint8_t  core_address;
        uint16_t hdata;                         // Operation specific data
        uint8_t  data_length;                   // data to follow, in 4 byte blocks, 0=no data
        uint8_t  crc8;                          // Computed across bytes 1-6 inclusive
        } __attribute__((packed,aligned(4)));   // 8 bytes total
```

where it is assumed that the underlying machine and compiler architecture were such that, if through a casting operation the structure were transmitted one byte at a time, it would be transmitted LS byte first, i.e. preamble would be the first byte sent, and it would be packed such that the total number of bytes sent would be eight. This assumption is only valid for common little endian CPU architectures in use today.

2.4.1 Privileged operations

Privileged operations are those which effect fundamental device behavior and should be protected against inadvertent manipulation.

2.4.1.1 OP_ROOT – Set to privileged mode

The GN ASIC powers up in "privileged" mode, so it is not normally necessary to perform this operation. However, at any time, the host may set a device into privileged mode by sending it an *OP_ROOT* packet. The host may place all devices in a chain in privileged mode by sending an *OP_ROOT* packet with the chip address set to the broadcast address.

When addressed to a specific target, the packet is discarded by that target. When addressed to the

broadcast address, the packet will be forwarded by all devices in the chain, and will arrive back at the host. If the host has a specific set of commands to perform privileged operations to either a particular target or all targets, these commands may in general be sent sequentially without any notion of "waiting" for the command to take effect, because commands take effect immediately upon reception. For example, if the host wanted to adjust the clock frequency for a particular device, it would send out a series of commands, bracketed by OP_ROOT/OP_UNROOT, to perform the operation. These commands may be sent back-to-back, because they are acted upon immediately upon validation of the header CRC.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_ROOT
chip_address	Target device, or 0xff for all devices
core_address	Unused
hdata	Unused
data_length	0 (No data, header only)

2.4.1.2 OP_RESET – Soft reset operation

The host may issue a soft reset to the GN ASIC. Although this may be directed at a specific ASIC, it is more likely that this operation will be used to reset the entire chain of ASICs. The operation is set up so that an ASIC will not initiate the soft reset until the packet has been fully forwarded on to the next device in the chain.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_RESET
chip_address	Target device, or 0xff for all devices
core_address	Reserved (Not used)
hdata	Reserved (Not used)
data_length	0 (No data, header only)

Note that this operation will revert serial baud rate to the default power-up rate. This operation may only be performed in privileged mode.

If this operation is directed to the broadcast address, then the host will receive the OP_RESET packet back, providing an indication that all devices are in the process of resetting. The host should wait a minimum of 100 usec after receiving back the OP_RESET packet before attempting to resume communications (at the default power-up baud rate) with the device chain.

2.4.1.3 OP_PLL_CONFIG – Configure PLL parameters

Hash engines use a high speed programmable clock. The host may re-configure this clock. During reconfiguration, all hash engines are reset, and any active hash job will be terminated. In the GN ASIC, the programmable clock derives its internal Voltage Controlled Oscillator (VCO) frequency by phase locking it to a 125 Mhz reference clock, or a divided value of this reference clock. The VCO operating range is 2000 – 4000 Mhz. The high VCO frequency is then scaled down to produce the hash clock. The PLL may be bypassed, allowing the 125 Mhz reference clock to be used as the hash clock. OP_PLL_CONFIG is a privileged operation, and will have no effect in non-privileged mode.

The following figure shows the PLL arrangement:

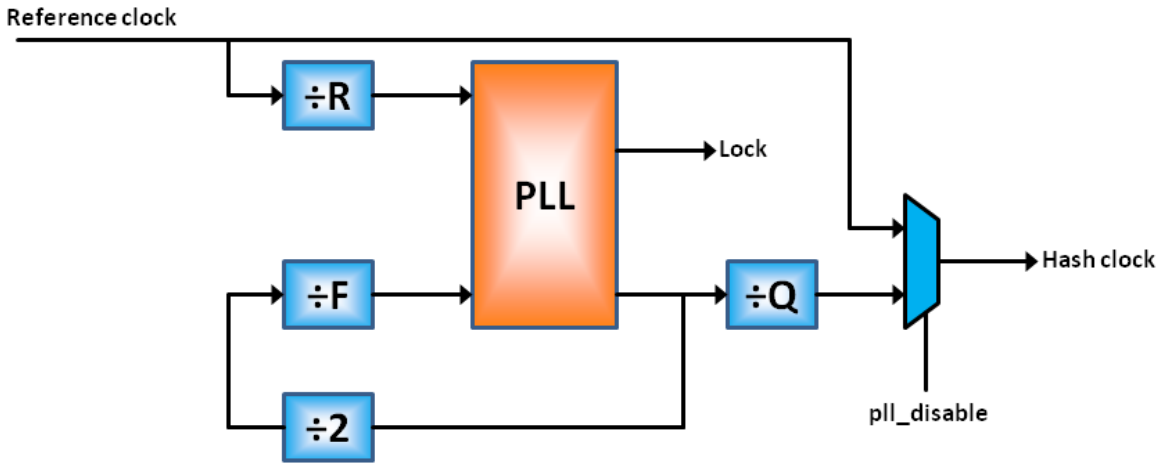


Figure 2.7: GN ASIC PLL Arrangement

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_PLL_CONFIG
chip_address	Target device, or 0xff for all devices
core_address	pll_config[23:16]
hdata	pll_config[15:0]
data_length	0 (No data, header only)

A 24 bit PLL configuration word is formed by concatenating data in the (unused) core_address field and the hdata field. The 24 bit PLL configuration word consists of the following fields:

Field	Value/meaning (host to device)
pll_config[23]	PLL_RESET. A one-shot bit that the host sets whenever PLL reconfiguration is performed. The host does not need to clear the reset condition, it is cleared automatically by the hardware.
pll_config[22]	PLL_BYPASS. If set, the PLL is disabled, all of the following fields are don't cares, and hash cores use the external 125 Mhz reference clock.
pll_config[21:16]	PLL_DIVR. Reference divider value. The external 125 Mhz reference clock can be divided to form the actual reference value used for the PLL. A value of 0 means no division (reference is 125 Mhz), 1 means divide by 2 (reference is 62.5 Mhz), etc.
pll_config[15]	PLL_FSE. Internal/External feedback. Must always be set to 1 (internal feedback).
pll_config[13:12]	PLL_RANGE. This 3 bit field sets the PLL filter range, which depends on the internal reference value (as set by PLL_DIVR). The possible values are: 000 = BYPASS 001 = 10-16 Mhz 010 = 16-26 Mhz 011 = 26-42 Mhz 100 = 42-65 Mhz 101 = 65-104 Mhz 110 = 104-166 Mhz 111 = 166-200 Mhz
pll_config[11:9]	PLL_DIVQ. PLL output divider value, the amount by which the internal VCO frequency is divided by to derive the actual hash clock rate. Allowed values are: 001 = Divide by 2 010 = Divide by 4 011 = Divide by 8 100 = Divide by 16 101 = Divide by 32 110 = Divide by 64
pll_config[8]	Reserved, set to zero.
pll_config[7:0]	PLL_DIVF. Feedback divider value. The VCO frequency is always divided by 2. The net frequency is then divided by a factor controlled by the PLL_DIVF field. 0 = divide by 1, 1 = divide by 2, 2 = divide by 3 etc.

Example:

To set the hash clock rate to 500 Mhz:

Use an internal VCO frequency of 2000 Mhz. We need the output divider set to 4, so PLL_DIVQ = 010. We set the reference clock to 125 Mhz, with PLL_DIVR=0 so there is no reference division. We need the VCO frequency divided by 16 to match this reference value. Allowing for the internal x2 division, PLL_DIVF needs to be set to 7, corresponding to a division of 8. PLL_BYPASS needs to be zero, and since the reference frequency is 125 Mhz, the PLL_RANGE needs to be 110.

The final pll_config value is made up as (in binary):

{1,0,000000} {1,110,010,0,00000111}

so in the OP_PLL_CONFIG packet, we set the core_address field to 0x80, and the hdata field to 0xe407.

No protection exists against illegal value combinations for the OP_PLL_CONFIG packet. It is the user's responsibility to use settings that are self consistent, maintain the VCO within its specified operating range, and maintain the hash clock within its specified operating limits.

2.4.1.4 OP_ADDRESS – Address cycle

NOTE: All systems that deploy more than one device in a chain MUST have this command issued immediately after power-up.

An address cycle is a mandatory operation that configures the "chip address" for each device in a chain. The host sends this packet, with the "chip_address" normally set to 0. When this packet is received, an ASIC sets its own "chip address" to the address in the packet, and increments the address in the forwarded packet.[3]

As a result, ASICs set their addresses to a consecutive range of numbers, 0, 1, 2 etc. for as many devices as there are in the chain. When the host receives back an OP_ADDRESS packet, the "chip address" field of the packet contains a count of the number of ASICs in the system.

The host should maintain a timer, and resend the OP_ADDRESS packet if it does not receive a response back within a reasonable time. Multiple sequential failures will indicate a problem in the hardware chain.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_ADDRESS
chip_address	0
core_address	Unused
hdata	Unused
data_length	0 (No data, header only)

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_ADDRESS
chip_address	Number of devices present.
core_address	# of cores in each device
hdata	Device ID in b7:0, Reference clock rate (Mhz) in b15:8
data_length	0 (No data, header only)

Several useful items are returned in the *hdata* field. The LS byte contains the ASIC revision number. The MS byte contain the reference clock value, in MHz. This reference clock value is used to compute new values of baud delay for an *OP_BAUD* packet, if the host wishes to change the baud rate. Values

3 Forwarded packets always have their header CRC re-generated, so it is OK to modify the header in transit.

of device ID are:

Device ID	Description
0	Reserved
1	Hashfast GN ASIC, Reference clock rate = 125 Ghz, Hash divisor = 1
128	Xilinx VC709, Hash clock rate = 250 Mhz, Hash divisor = 1

2.4.1.5 OP_READDRESS

Note: This command deprecated in GN

As outlined in section 2.2.1, there may be reasons for the host to readdress individual cores. This command may be sent, in privileged mode, in order to bring about this readdressing.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_READDRESS
chip_address	Target device, or 0xff for all devices
core_address	Existing core address – CANNOT be broadcast address
hdata	New core address in b7:0
data_length	0 (No data, header only)

2.4.1.6 OP_HIGHEST

Note: This command deprecated

As a result of readdressing, the total number of cores the host considers the device to have may be lower than the physical core count. This command may be used to adjust the device's notion of its actual core count, for the purpose of status reporting.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_HIGHEST
chip_address	Target device, or 0xff for all devices
core_address	Unused
hdata	Highest desired core address (must be > 0)
data_length	0 (No data, header only)

2.4.1.7 OP_BAUD

Dynamic baud rate changing is a tricky operation. The host may reprogram the baud rates for all devices in a chain, but only under circumstances where devices cannot produce packets independently, i.e. no hashing should be active while this operation takes place, and periodic status frame generation should be disabled. This is not an unreasonable limitation, any baud rate change would typically take place only once during system initialization.

After the host sends this command, it should not send any subsequent commands until it receives an OP_BAUD response back, indicating that all ASICs in the chain have changed their baud rate. This response will come back *at the old baud rate*.

Internally, each device changes its baud rate *only after the entire OP_BAUD packet has been transmitted on to the next device (*at the old baud rate), or for the last device on to the host. The host must leave its interface baud rate at the existing/old setting until it receives the OP_BAUD reply, and only after that time can the host change its baud rate to correspond to the new rate.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_BAUD
chip_address	Must be 0xff for all devices
core_address	Watchdog timeout to use around baud rate setting operation (LS 7 bits) (seconds).
hdata	New baud delay (see below)
data_length	0 (No data, header only)

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_BAUD
chip_address	0xff
core_address	Unused
hdata	Baud delay
data_length	0 (No data, header only)

The host does not send a “baud rate” in this command, it sends a “baud delay”, based on the reference clock rate used by the ASIC. This reference clock rate is given to the host in the response header to the mandatory OP_ADDRESS cycle.[4] The baud delay is computed by the host as:

4 This reference clock rate is a number built into the ASIC design. It is entirely possible for someone to make a board level product which feeds a *different* reference clock rate into the ASIC. In this situation, the host will need help – knowledge from some other authority – to use a proper reference clock rate for deriving baud delay, and will need to ignore the reference clock rate returned in the OP_ADDRESS response.

$$baud\ delay\ =\ \frac{reference\ \ clock\ \ frequency\ \ (Mhz)\ \times 10^{6}}{baud\ rate} - 1$$

For example, if the reference clock frequency was 125 Mhz, and the desired baud rate was 115200, then the baud delay would be set to 1084 (rounding to nearest integer).

The baud delay may not exceed 65,535. Practically speaking, for a 125 Mhz reference clock rate, this precludes standard baud rates of 1200 and lower. The maximum available baud rate depends upon the reference clock and switching characteristics of the board level circuitry. For a 125 Mhz reference clock, baud rates above 2 Mbaud are not recommended, and hardly necessary.

The host can specify a “watchdog timeout”, to be initiated at the same time the baud rate is changed on each device. Internally, this occurs just after the OP_BAUD packet has been fully transmitted (forwarded) to the next device in the chain, or in the case of the last device, back to the host. Once the host receives the OP_BAUD packet back, it must change its own baud rate to reflect the new value. The host must then send any kind of packet through the device chain, in order to reset the watchdog timer. If the GN ASIC does not receive a package within the specified watchdog timeout after an OP_BAUD operation, then a soft reset will ensue and the device will revert back to the default (power-up) baud rate, specified by the values of gpi[7:5] during reset. Recommended behavior would be for the host to specify a relatively short (e.g. 3 seconds) watchdog timeout value in the OP_BAUD operation, and to reset the watchdog timeout to a longer value (e.g. 30 seconds) using an OP_CONFIG operation, as one of the first operations to be done at the new baud rate.

2.4.1.8 OP_CLOCKGATE

The GN ASIC has provisions for gating the hash (high speed) clock "off" for individual search blocks. This allows some power recovery from blocks which are not being used, due to work starvation or thermal limitations. The default state for all clock gates is "on", so after power-up or a soft reset, all hash cores will be enabled.

The form of this packet depends on the number of hash cores present. If there are less than or equal to 16 cores, then the packet is a "short" packet and the clock gating bitmap is contained in the "hdata" field. If there are more than 16 cores (e.g. In the case of the GN ASIC there are 96 cores), then this is a "long" packet, and the body data consists of a bitmap to be used to specify the core enable map. Core 0 corresponds to the LS bit of the first byte. Core 9 corresponds to the LS bit of the second byte, etc.

Here are the packet fields for the GN ASIC:

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_CLOCKGATE
chip_address	Target device, or broadcast address
core_address	Unused, reserved
hdata	Unused, reserved (since there are > 16 hash cores)
data_length	3 (corresponding to 12 bytes of data to follow. Does not include CRC-32 trailer).
DATA	12 bytes of bitmap

2.4.1.9 OP_UNROOT

When the host has done all of the privileged configuration operations, it may send an OP_UNROOT command to take one or all devices out of privileged mode, to prevent future inadvertent manipulation of important configuration information. This operation is entirely optional – there is no reason the host cannot leave all devices in a privileged state. Hash processing will be able to occur regardless.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_UNROOT
chip_address	Target device, or 0xff for all devices
core_address	Unused
hdata	Unused
data_length	0 (No data, header only)

A response will not be received, unless the chip_address is set to 0xff, in which case the response received will be identical to the above command sent out.

2.4.2 Non privileged (user mode) operations

2.4.2.1 OP_HASH Queue a hash job

This is the main command used by the host to initiate a hash search. After the system is initialized, most of the host's activity will consist of procuring work from a mining server, and farming out work to every hash core in every ASIC, to initiate as much parallel searching as possible.

The OP_HASH command has various capabilities to allow tailoring the search process to suit different use scenarios. Some examples of different use scenarios are:

- A solo mining effort may distribute the same block header to many cores, with different base timestamp values, or different midstates by lieu of different extranonces, and iterate through many 2^{32} nonce sweeps in the hope of finding a solution. Search difficulty would be set close to the actual required difficulty for the block, so very few candidate solutions would be passed back to the host. The job queuing traffic would be very low. However, when a proper solution was found, all operations would be halted, and new jobs set up across the entire engine.
- A pool miner would secure jobs from the mining pool, and queue two jobs up per hash core, so that there was always a pending job to take over when the active job completed. As each job completes, the host would queue a new pending job to that core. If jobs go stale, the host would send abort directives to end those jobs, along with replacement, fresh, hash jobs. Difficulty level would be set at the required pool mining levels, so fairly frequent candidate solutions would be found.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_HASH
chip_address	Target device
core_address	Target core
hdata	Sequence number of hash job
data_length	15 (corresponding to 60 bytes of data to follow. Does not include CRC-32 trailer).
DATA	See below

Considered as a packed C data structure on a little endian architecture, the format of the packet data frame sent to initiate a hash search is:[5]

```
struct hf_hash_serial {
        uint8_t  midstate[32];                  // Computed from first half of block header
        uint8_t  merkle_residual[4];            // From block header
        uint32_t timestamp;                     // From block header
        uint32_t bits;                          // Actual difficulty target for block header
        uint32_t starting_nonce;                // Usually set to 0
        uint32_t nonce_loops;                   // How many nonces to search, or 0 for 2^32
        uint16_t ntime_loops:12;                // How many times to roll timestamp. Set to 0 for G-1.
        uint16_t spare1:4;
```

5 With apologies to Big Endian users, and to all users who are compiler purists about the use of bit fields.

```
uint8_t  search_difficulty;            // Search difficulty to use, # of leading '0' bits
uint8_t  options;                      // Configuration options
uint32_t group_id:8;                   // Group ID if options[0] is set
uint32_t spare2:24
uint32_t crc32;                        // Computed across all preceding data fields
} __attribute__((packed,aligned(4)));  // 64 bytes total, including CRC
```

The total packet length, then is 72 bytes, consisting of an 8 byte header, followed by 60 bytes of data and a 4 byte CRC-32 check field. The following notes describe certain fields in more detail:

hdata (header field)

The header *hdata* field for OP_HASH is a 16 bit sequence number chosen by the host. The number has no meaning to the hash core, but it is returned along with candidate nonces in OP_NONCE packets which represent hits from the search process. By using this sequence number, the host can determine which search job the candidate nonce goes with!

The host would usually elect to to roll sequence numbers across a smaller range, depending on how many hash cores are in the entire chain. For example, if there are 24 ASICs in the chain, and each ASIC contains 32 hash cores, there will be as many as 24*32*2 = 1,536 search jobs going on concurrently. The host can elect to track these by using 11 bits of the sequence number space. The MS 5 bits of the sequence number field can remain unused, or the host can use this space for additional context information. This is completely up to the host. All 16 bits will be returned to the host.

midstate (32 byte data field)

This is the midstate, that would traditionally be returned to mining applications in a *getwork* request, which has more recently been deprecated because miners can easily calculate it from the block data. It represents the SHA-256 internal state after processing the first 512 bits of the block header, which does not change across search operations.

merkle_residual (4 byte data field)

timestamp (4 byte data field)

bits (4 byte data field)

These three fields represent the first 96 bits of the second half of the block header, and do not change across a 2^32 iteration nonce search. However, if the user elects to have ntime rolling performed as part of the search process, then the timestamp value will automatically be incremented by the hardware in successive nonce search iterations, to the limit determined by the *ntime_loops* field.[6]

starting_nonce (4 byte data field)

This field will normally be zero. Allowing it to be set to other values is to facilitate testing and particularly simulation work.

nonce_loops (4 byte data field)

6 ntime rolling at the ASIC level is unavailable in the G-1 stepping

This field specifies how many search iterations to perform, in the range 1 .. 2^{32}. A value of zero will normally be set, which corresponds to searching through the full 2^{32} range.

ntime_loops (12 bit data field)

This field specifies how many additional search sweeps to perform through the mechanism of ntime rolling, after the initial search has completed. If set to zero, no ntime rolling will be performed. If set to a non-zero value, ntime rolling will be performed. If this is the case, it makes no sense to set nonce_loops to anything other than zero (so each nonce sweep represents 2^{32} tries). If this field is set to a large number, the hash core could potentially spend a very long time before it completes the search. The host always has the option of halting the search (through an *OP_ABORT* operation) if it decides that the work being operated on has gone stale.[7]

search_difficulty (1 byte data field)

This field is used to select candidate nonces for passing back to the host. It specifies how many leading '0' bits are required in the nonces that constitute search output results. This field allows tailoring the search process to suit the application, e.g. mining pools vs. solo mining, and to select appropriate nonce values for further examination by the host without leading to too many false positive indications.

This field must not be zero. A setting of 32 corresponds to “Diff 1”; using any value below this is pointless for today's mining applications. The maximum value it can take is 128 for the GN ASIC. This corresponds to a Bitcoin Difficulty of $7.922 \text{x} 10^{28}$ which is an inconceivably high level. For a single GN ASIC hashing at a rate of 400 GH/sec, it would take $2.6976 \, x \, 10^{19}$ years, on average, to find such a nonce. This is 1,955,044,766 times the generally accepted age of the Universe. HashFast's ASIC developers are working feverishly on the next generation ASIC to reduce this number!

No direct response occurs to the *OP_HASH* packet, unless it is addressed to the broadcast chip address. This would normally only be done for testing, or for group operations (see later). If a CRC-32 error occurs, the search job will not be queued to the target core. No advice about this is sent to the host – it should be an extremely rare event – but the host will notice the job as not having been launched during subsequent activity notifications (sequence numbers will simply roll past it), and can replace it.

There may, of course, be one or more OP_NONCE packets directed to the host, which can contain candidate nonces as a result of the requested operation. These only occur as hits are obtained according to *search_difficulty* – and it may often be the case that no hits occur.

At 1,152,000 baud, the 72 byte OP_HASH packet takes 625 usec for transmission, which means a maximum of 1,600 jobs per second across the system. If there are so many cores available that this rate is inadequate, then the baud rate may be increased to accommodate. For instance, setting the rate to 3 Mbaud would allow 4,167 jobs per second – way beyond the typical capabilities of a single chain doing full span searches.

7 Ditto. Always set the ntime_loops field to zero for the G-1 stepping.

2.4.2.2 OP_ABORT – Abort a hash search

Search operations may be configured to take a significant amount of real time. For one reason or another, the host may wish to abort either the currently active search job, and/or the pending search job. A job becoming 'stale', for example, would cause the host to want to terminate an active search.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_ABORT
chip_address	Target device, or 0xff for broadcast address
core_address	Target core, or 0xff for all cores
hdata	b0: Abort current search job, b1: abort pending search job b2: Abort by group_id code in b15:8 b15-8: Group code to abort if b2 is set
data_length	0 (No data, header only)

No reply will be made to this command, if it is directed at a specific device. If it directed to the broadcast address, a response will be received, in the form of the host's own packet returning.

Note that if the host wishes to abort all searches in all cores of all devices, it can achieve this by sending a single *OP_ABORT* command, with both the *chip_address* and *core_address* set to 0xff, and the *"hdata"* field set to 0x3.[8]

2.4.2.3 OP_NONCE – Returned candidate nonce(s)

This command is never initiated by the host. It is only ever initiated by a device, when candidate nonces are available for returning to the host. Candidate nonces are those found to satisfy the *search_difficulty* criteria in *OP_HASH* search operations.

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_NONCE
chip_address	The address of the chip sending this notification
core_address	Unused
hdata	Unused
data_length	Variable, always non-zero
DATA	One or more structures as described below

8 Hardware reasons may make this a bad idea.

Recall from the discussion around Figure 2.6 how candidate nonces are multiplexed into a single queue. These events will generally be infrequent, however traffic conditions and lowered search difficulty levels may lead to more than one nonce to be pending for transmittal to the host.

An OP_NONCE packet will contain as many candidate nonce structures as there are available at the time the packet was constructed. The length of the data frame can be used by the host to determine how many nonces have been returned. Each returned nonce will consist of 8 bytes, packed in accordance with the following structure:

```
struct hf_candidate_nonce {
        uint32_t nonce;                         // Candidate nonce
        uint16_t sequence;                      // Sequence number from corresponding OP_HASH
        uint16_t ntime:12;                      // ntime offset, if ntime roll occurred
        uint16_t search:1;                      // Search forward 128 values to find more nonce(s)
        uint16_t spare:3;
        } __attribute__((packed,aligned(4)));
```

The host can use the sequence number to match this nonce to the work item it corresponds to. If ntime rolling was employed, the returned ntime field contains the ntime offset that must be applied to the work for this nonce.

Occasionally, two nonces may be found in close proximity to each other. Due to hash engine pipelining, it is awkward to return multiple nonces within a certain proximity. What, for example, should the hardware do in the inconceivable case of six successive nonce values? In order to alleviate this situation, we return the proximity flag *"search"* in the above structure. If set, then if the host cares to search forward (in software) through the next 128 successive nonce values, then it is guaranteed to find *at least* one other qualified nonce.

2.4.2.4 OP_STATUS – Status notifications

The host needs to know when search jobs complete, so that new search jobs can be queued to the relevant cores. Recall each core can support one active search job, and one pending search job. Whenever an active search job completes in a core, an OP_STATUS notification may be scheduled and sent after a configurable batching delay. In addition, the host may configure the ASIC to periodically send out OP_STATUS packets, regardless of whether or not trigger conditions have occurred.[9]

The status of all cores is sent back in OP_STATUS packets. The batching delay would typically be set to a relatively large time – 200 msec, for instance, to reduce the frequency of OP_STATUS notifications to just a few per second. Recall that each hash core can have one active job and one pending job. Search operations will normally take several seconds, giving plenty of time for the host to queue a fresh job to each core that has an empty pending job slot.

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_STATUS
chip_address	The address of the chip sending this notification

9 This in fact is the most driver friendly mechanism, and the one generally used.

Field	Value/meaning (host to device)
core_address	b7 = Thermal cutoff happened, b6:5=unused, b4:0=tacho_csec
hdata	Most recently received OP_HASH sequence number before this status message was generated
data_length	0 if there are 16 cores or less, or non zero for more core data
DATA	Four 16 bit monitoring values, followed by a bitmap of core status, 2 bits per core, corresponding to {pending,active} status.

The inherent delay associated with serial communication can lead to coherency problems when interpreting OP_STATUS messages. For example, a host user mode application has issued a write() which queues an OP_HASH to a given core, but then immediately reads an OP_STATUS reply which indicates the core is inactive. Has the OP_HASH completed without finding any nonces? Or is it still largely held in the host's transmit buffer, and hasn't even made it out to the device yet?

In order to resolve these situations, a sequence numbering system is used. The host includes a sequence number in each OP_HASH packet, which would normally be incremented for each hash job. OP_STATUS packets contain a sequence number in the “hdata” field, which corresponds to the most recent OP_HASH sequence number received and fully queued to its respective core. As a result, the host can compare the received OP_STATUS sequence number with that which was sent for the active and/or pending job on a given core, and discount the OP_STATUS activity bits if in fact the sequence number reflects data which does not yet include the most recently queued OP_HASH jobs.

Sequence numbering allows arbitrary delays in the serial communication, in addition to natural delays which might for instance occur to a user-mode mining application on a heavily loaded system.

There is an inherent robustness associated with the sequence numbering scheme and the associated periodic status notifications. If an OP_STATUS packet goes missing (in the unlikely event of a communications error), the same information will be contained in a subsequent OP_STATUS packet, so the host will not lose any capability as a result of the transmission loss. Similarly, if an OP_HASH went missing because of a communications error, the host would know it went missing once the sequence numbering rolled past the missing work, and as a result the host would generate a replacement.

The 64 bits of monitoring values returned at the beginning of the data segment are system specific, as is the conversion between each the embedded numbers and the associated floating point value. In the case of the GN ASIC, the monitoring values are as follows:

```
struct hf_g1_monitor {
        uint16_t die_temperature:12;                // Die temperature ADC count
        uint16_t spare:4;                           // Spare
        uint8_t  core_voltage_0;                    // Core voltage,main sensor position
        uint8_t  core_voltage_1;                    // Core voltage position A
        uint8_t  core_voltage_2;                    // Core voltage,position B
        uint8_t  core_voltage_3;                    // Core voltage,position C
        uint8_t  core_voltage_4;                    // Core voltage,position D
        uint8_t  core_voltage_5;                    // Core voltage,position E
        } __attribute__((packed,aligned(4)));
```

The temperature conversion equation is:

$$T = (N * 240) / 4096 - 61.5$$

where N is the 12 bit “die_temperature” field from the above structure, and T is the temperature in degrees Celsius.

The voltage conversion equation is:

$$V = N / 256 * 1.2$$

where N is an 8 bit “core_voltage” field from the above structure, and V is the measured voltage in Volts. The GN ASIC has 9 voltage measurement positions. The “main sensor” position is always measured, and returned in the core_voltage[0] field above. Of the eight remaining monitoring points, the user may choose any five to be returned in the core_voltage[1] to [5] fields above. Which five are monitored is specified by an 8 bit register (voltage_sample_points) included in the OP_CONFIG register write operation, with bits set corresponding to the monitoring positions to be returned.

If the tachometer option is selected (see OP_CONFIG, hdata field), the monitoring values are modified as follows:

```
struct hf_g1_tachmon {
        uint16_t die_temperature:12;            // Die temperature ADC count
        uint16_t spare:4;                       // Unused
        uint8_t  core_voltage_0;                // Core voltage,main sensor position
        uint8_t  core_voltage_1;                // Core voltage position A
        uint8_t  core_voltage_2;                // Core voltage,position B
        uint8_t  core_voltage_3;                // Core voltage,position C
        uint16_t tacho_count;                   // Tachometer accumulation count
        } __attribute__((packed,aligned(4)));
```

The “tacho_count” field is the number of tachometer impulses measured on the gpi[3] input across a certain number of 0.1 second intervals. Just how many intervals is specified in b4:0 of the (unused) core_address field of the packet header (marked tacho_csec in the above header fields table). Implicit in use of the tachometer is that status messages must be sent from each device to the host at least every 3.1 seconds. A usual configuration would have status messages going back to the host several times per second.

Example: if the “tacho_count” value is 980, and the “tacho_csec” value is 4, then the tachometer reading is 980/0.4 = 2,450 rpm.

If the tachometer option is employed, the number of voltage sample points (in addition to the main sensor) that may be returned in an OP_STATUS message is reduced from five to three. This means that a maximum of 3 bits may be set in the 9 bit voltage_sample_points register if the tachometer option is employed.

2.4.2.5 OP_GPIO – General purpose I/O

The Hashfast ASIC includes 8 general purpose inputs and 8 general purpose outputs. Outputs will power-up disabled, and the host may enable any combination of these to perform system specific tasks, if desired. General purpose inputs are read, and outputs are written, by the OP_GPIO packet.

In addition to simple read/write of individual pins, there are certain built-in capabilities for the general purpose output pins:

- Various built-in output strobing capabilities exist, these may be used directly for tasks such as LED flashing.
- Up to three output pins may be dedicated to a 1, 2 or 3 phase PWM output capability, which can form a simple interface for peripherals such as fans which use such control methods

Apart from basic PWM configuration setup, which is done through registers set up with the OP_CONFIG, command, all output functions are controlled through the OP_GPIO command.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_GPIO
chip_address	Target device, or 0xff for broadcast address
core_address	Various functions - see table
hdata	GPIO or PWM output settings – see table
data_length	0 (No data, header only)

Not all I/O may be bonded, but the programming model is based on 8 general purpose inputs and 8 general purpose outputs. These correspond with device ports gpi[7:0] and gpo[7:0] respectively.[10]

General purpose input pins 5, 6 and 7 are reserved during power-up to set the default baud rate. After power-up or a hard reset, these pins may be used for any purpose. The default baud rates set during power-up or a hard reset are as follows:

B7:5 value	Baud rate
0	115,200
1	9,600
2	38,400
3	57,600
4	230,400
5	576,000

10 These ports are also used as a byte wide I/O interface for simulation and testing, but this capability will not typically be used in practice.

B7:5 value	Baud rate
6	921,600
7	1,152,000

The following table summarizes OP_GPIO operations, as controlled by the value in the core_address field. This field, of course, does not refer to any particular core for the OP_GPIO operation.

core_address value (b2:0)	Operation, and meaning of hdata field
0	Set output enables and values, hdata[7:0] = values, hdata[15:8] = active high enables
1	Read inputs only, and return values in hdata[7:0]
2	Set internal LED blink mask from hdata[7:0], LED persist time (msec*16) in hdata[11:8], PWM enable mask in hdata[14:12] (which overrides internal LED blink)
3	PWM level in the range 0 – pwm_pulse_period (see OP_CONFIG). Bit 7 of the core_address in this case is an active high enable for PWM outputs.

The 3 bit operation code in core_address bits 2:0 control the nature of the OP_GPIO operation.

For an operation code of 0, the OP_GPIO sets output pins directly. In the hdata header field, the LS byte contains the values to be written to each of the 8 output pins, and the MS byte contains an active high mask, with a bit set for each output pin indicating which outputs should be enabled. There is no partial write capability, the host can easily emulate such behavior by maintaining an internal copy of the last written value.

For an operation code of 1, no output change occurs. The eight general purpose inputs will be read, and their value will be returned to the host in the LS byte of the hdata field in a response packet forwarded to the host.

The gpi values will also be returned to the host in the LS byte of the hdata field for an operation code of 0, if the destination chip address in the packet is specifically addressed to a single device. If the destination chip address is in fact the broadcast address, then gpi values will not be read, and instead the hdata output setting will stay the same and gpo pins on all devices will be set using the same information.

An operation code of 2 allows overriding the normal behavior of output ports, and instead tying them to internal strobing features and/or PWM outputs. The LS byte of the hdata field contains an 8 bit mask which enables up to 8 internally generated strobes with various meanings (described below). Bits 11:8 of the hdata field contains a “persistence” value, indicating how long each strobe will remain high once triggered, in units of 16 msec increments. This field is typically used to specify a persistence time for allow an external LED to “flash” for a period long enough to be visible. Finally, bits 14:12 of the hdata field specify which of up to 3 output ports are to be used as PWM outputs, over-riding all other uses for those output ports (gpo[2:0]).

An operation code of 3 specifies the PWM level, within the range 0 to 100%, for the PWM output system. The 100% level is specified by setting the hdata value to the same value as used for

pwm_pulse_period during the initial setup of the PWM parameters in the OP_CONFIG command.

Note that although there are three PWM outputs they are not independent – they may be used to set up a single PWM output, a 2-phase PWM system, or a 3-phase PWM system. There is only one PWM output “level” for any of these arrangements.

Finally, the following table summarizes the functionality for each output, if the LED flashing option is selected for that output. Note that these outputs are per-die. There are four die on the GN ASIC. Not also that if pwm outputs are selected, then the corresponding LED flashing outputs are not available since they share the same pins.

Output	**Operation**
gpo[7]	Heartbeat. Alternates ON or OFF once per second.
gpo[6]	Blinks ON for every packet received
gpo[5]	Blinks ON for every packet sent
gpo[4]	Blinks ON whenever a core goes from busy to non-busy (hopefully never happens!)
gpo[3]	Blinks ON for every nonce found
gpo[2]	Blinks ON whenever a header CRC error occurs
gpo[1]	Blinks ON whenever a receive timeout occurs
gpo[0]	Blinks ON for every packet forwarded

2.4.2.6 OP_CONFIG – Set configuration registers

Host drivers may issue an OP_CONFIG command to set up various configuration registers. These would typically be set only once during system initialization; however, the driver may alter them at any time. No response is sent back for this command, unless the target device is the broadcast address in which case the host will get its own message back.

Host sends:

Field	Value/Meaning
operation_code	OP_CONFIG
chip_address	Target device, or 0xff for broadcast address
core_address	Reserved
hdata	b15 = Write config registers, b14=Enable thermal overload limit, b13=Enable tacho b9:0 = Thermal overload limit if enabled (via b14).
data_length	4
	Configuration register data (see below)

The following C data structure describes the layout of the data segment, mapped into the associated internal registers. All fields are written – there is no selective write capability. The host can implement a modify capability by simply remembering what it last wrote.

```
struct hf_config_data {
        uint16_t status_period:11;                  // Periodic status time, msec
        uint16_t enable_periodic_status:1;          // Send periodic status
        uint16_t send_status_on_core_idle:1;        // Schedule status when a core goes idle
        uint16_t send_status_on_pending_empty:1;    // Schedule status when a core pending goes idle
        uint16_t pwm_active_level:1;                // Active level of PWM outputs
        uint16_t forward_all_privileged_packets:1;  // Forward priv pkts - diagnostic
        uint8_t  status_batch_delay;                // Batching delay, msec
        uint8_t  watchdog:7;                        // Watchdog timeout, seconds
        uint8_t  disable_sensors:1;                 // For diagnostics

        uint8_t  rx_header_timeout:7;               // Header timeout in char times
        uint8_t  rx_ignore_header_crc:1;            // Ignore rx header crc's (diagnostic)
        uint8_t  rx_data_timeout:7;                 // Data timeout in char times / 16
        uint8_t  rx_ignore_data_crc:1;              // Ignore rx data crc's (diagnostic)
        uint8_t  statistics_interval:7;             // Stats reporting interval in seconds
        uint8_t  stat_diagnostic:1;                 // Diagnostic use only, set to 0.
        uint8_t  measure_interval;                  // Die temperature measurement interval in msec

        uint32_t one_usec:12;                       // How many LF clocks per usec – 1.
        uint32_t max_nonces_per_frame:4;            // Maximum # of nonces to send in a single frame
        uint32_t voltage_sample_points:8;           // Bitmap of voltage sample points (max 5)
        uint32_t pwm_phases:2;                      // Number of PWM phases – 1
        uint32_t trim:4;                            // Trim value for temperature measurements
        uint32_t clock_diagnostic:1;                // Diagnostic use only, set to 0
        uint32_t forward_all_packets:1;             // Forward everything - diagnostic
```

```
uint16_t pwm_period;                    // Basic period per PWM phase
uint16_t pwm_pulse_period;              // PWM cycle width
} __attribute__((packed,aligned(4)));
```

The “hdata” field controls several optional components. In order to allow independent control of these optional components, bit 15 of the “hdata” field controls whether or not the register data contained within the packet body is to be written or not. Bit 15 must be set for the register data to be written.

Bit 14 is an enable for the “thermal overload limit” monitor. If set, bits 9-0 of the hdata field contain a raw temperature monitor count which represents the absolute upper limit on die temperature. If the measured die temperature exceeds this limit, then a “thermal shutdown” event will occur. The following actions are taken in the event of a thermal shutdown:

- All hash engines are reset, and the PLL is placed in bypass mode
- All future OP_STATUS messages sent to the host will have bit7 of the “core address” field set, indicating to the host that thermal cutoff happened. This bit will stay set until the host issues an OP_PLL_CONFIG operation to re-configure the PLL and bring hash engines back out of reset.
- The gpo[7] one second flashing heartbeat, if enabled by the user, will change to flash three times per second. The flash rate will revert to once per second after the host next issues an OP_PLL_CONFIG operation.

Thermal shutdown is an extreme event, and should not normally occur. The host driver should operate the GN ASIC in such a manner that die temperature is controlled within limits set below the “thermal overload limit”. There may be cases where the host driver does not perform this function well enough, such as if the host crashes immediately after setting large/lengthy jobs in place. Protection against this type of unexpected situation is the main purpose of the thermal overload feature.

Bit 13, if set, enables a “tachometer” feature, associated with gpi[3].

Note regarding OP_STATUS generation

There are several options in the above data structure that control how the important OP_STATUS messages are generated. These messages are the only means by which a controlling device can determine the status of cores. In more detail, the fields controlling OP_STATUS generation are:

- *status_period* – The period, in msec, between generation of OP_STATUS messages, if *enable_periodic_status* is set.
- *enable_periodic_status* – If enabled (set to 1), OP_STATUS messages will be generated periodically, regardless of core activity.
- *send_status_on_core_idle* – If any core goes idle, generate an OP_STATUS message, after the batching delay if configured.
- *send_status_on_core_pending* – If any core goes pending, generate an OP_STATUS message, after the batching delay if configured.
- *status_batch_delay* – If either of the above two options are configured, and are triggered, then start a timer and wait until that timer expires before actually generating the OP_STATUS. This

is to limit the rate of OP_STATUS generation, by picking up other core active/idle events during the “batching period” before these are sent to the host.

In general, the least useful of these options is *send_status_on_core_idle*. Recall that hash cores are double buffered, there is a pending job behind the active hash job, ready to commence as soon as the active hash job has completed. The host's ambition should be to keep *pending* slots full. If the host does this, the active core jobs will take care of themselves.

HashFast recommends using only the *periodic* status notification methods. Useful thermal data is passed back to the host in OP_STATUS messages, and the host should monitor this regularly. Since hash jobs can be aborted, there is generally no reason to limit nonce ranges, so if hash jobs run across the full 2^{32} nonce range, each jobs takes 3+ seconds to complete. As such, receiving a status update (say) every half second provides the host with ample time to notice an empty pending slot and fill it.

2.4.2.7 OP_GROUP– Set up group operations

Host drivers may set up “group” operations, where multiple cores (possibly across multiple ASICs) operating on the same hash job, with unique per-core adjustments to either nonce range and/or ntime rolling offset[11]. Any given core can only belong to one group at one time. Group ID's are 8 bit non-zero identifiers.

To use this facility, the host would typically issue a series of OP_GROUP packets, each with unique chip/core values, and with data modifiers, to set up each core's responsibilities in the group. Once set up, the host can issue a single OP_HASH command, directed to the broadcast core address, with the option field group mode set, and the group_id specified. Cores with matching group_id's will each pick up the job and go to work on their part.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_GROUP
chip_address	Target device
core_address	Target core
hdata	group_id in b7:0 If bit8 is set: Set group membership according to the group_id If bit8 is clear: Clear group membership
data_length	1
	Group configuration data (see below)

The following C data structure describes the layout of the data segment, mapped into the associated fields.

```
struct hf_group_data {
        uint16_t nonce_msoffset;                    // This value << 16 added to starting nonce
        uint16_t ntime_offset:12;                   // This value added to timestamp. Must be 0 for G-1
        uint16_t spare:4;
        } __attribute__((packed,aligned(4)));
```

11 ntime rolling is unavailable in the G-1 stepping

2.4.2.8 OP_STATISTICS – Statistics reporting

The GN ASIC maintains a small set of statistics registers, mainly to record anomalies and error rates.

The host does not poll these registers, they are reported to the host periodically so that the host can update a duplicate set of much wider counters that it maintains locally. The host sets the statistics reporting interval in the OP_CONFIG operation, as outlined in the previous section. After the reporting interval has expired, if any statistics counter has a non-zero value, an OP_STATISTICS frame will be generated, and the internal counters will be reset to zero. In general, there will be no errors and therefore no frame will be sent. If a frame is received, the host may add the reported statistics counters to an internal (wider) set of accumulated statistics.

The content of an OP_STATISTICS data frame is as follows:

```
struct hf_statistics {
        uint8_t rx_header_crc;                  // Header CRC's
        uint8_t rx_body_crc;                    // Data CRC's
        uint8_t rx_header_timeouts;             // Header timeouts
        uint8_t rx_body_timeouts;               // Data timeouts
        uint8_t core_nonce_fifo_full;           // Core nonce Q overrun events
        uint8_t array_nonce_fifo_full;          // System nonce Q overrun events
        uint8_t stats_overrun;                  // Overrun in statistics reporting
        uint8_t spare;
        } __attribute__((packed,aligned(4)));
```

Note that since the statistic counters are all for anomaly conditions, and that a packet is only sent if any counters are non-zero, it is quite possible for the host to never receive an OP_STATISTICS packet. This is normal behavior.

3 USB Communications Protocol

3.1 Introduction

Mining product deployments using the HashFast ASIC will often employ a USB interface as an intermediate communications mechanism between a chain of ASICs and a controlling CPU, even if that controlling CPU is an embedded processor in the mining equipment itself. Due to the reliability of USB connections, some of the serial protocol constructs are not required.

HashFast products which include a USB communications port allow all of the low level serial communications detailed in Chapter 1, but also provide a higher level set of system interfaces which provide the host with a simple, global work queue per device. These two protocols are for the most part mutually exclusive. A host will elect to use one or the other. The two protocols may be summarized as follows:

- USB Mapped Serial (UMS) protocol – Section 3.3

  Most of the protocol requests and responses described in Chapter 2 are available when using the USB interface. Data structures are almost entirely identical, except the CRC-32 check at the end of non-short packets is dropped[12], and a minor change is made to the OP_HASH structure so that the entire packet will fit into 64 bytes.

  This provides a low level interface, giving the host a great deal of control over how each hash core in each ASIC is used. However, to control this network, the host is potentially dealing with thousands of cores per device, and must maintain structures accordingly.

- USB Global Work Queue (GWQ) protocol – Section 3.4

  This protocol provides a higher level abstraction layer to the host, requiring only that the host send work to the device, and process nonces and status in a return queue. The low level details of how work is distributed is handled internally, and the host does not need to know these details. Work derivation techniques involving a small amount of ntime rolling may optionally be used internally, to dramatically lower the number of work items the host must generate, typically by a factor of 24 or 32. The host may still abort stale work through a protocol specific work restart mechanism, and work which is aborted may be immediately replaced with negligible loss of hash throughput.

  This protocol is easier for drivers to interface with, and with a little ntime rolling specified makes very little demand on CPU power. Modest CPU's can control large scale systems using the GWQ protocol. Unless there is a good reason to use the lower level protocol, new drivers should be written to use the GWQ protocol.

Section 3.3 of this Chapter details the USB mapped serial protocol, in the context of the underlying serial protocol it maps to as described in the previous Chapter. Section 3.4 of this Chapter describes the USB Global Work Queue protocol in detail.

12 Since USB communication is highly reliable

3.2 USB Vendor and Product ID.

HashFast products use the unique vendor ID:

0x279c

This vendor ID is assigned to HashFast by the USB Implementer's Forum (http://www.usbif.org) for a fee and cannot be used by others unless specifically sub-licensed from HashFast.

The Product ID for generation 1 GN based HashFast products is 0x0001. The USB 2.0 interface appears to the host as a Communications (CDC-ACM) device.

Early development emulators use a single 'Bulk' endpoint pair with endpoint max packet sizes (wMaxPacketSize) of 64 bytes. Later development emulators use two 'Bulk' endpoint pairs, one for the write direction and one for the read direction, with 512 byte max packet sizes.

3.3 USB Mapped Serial protocol

The same 8 byte header described in Chapter 2 is used to pre-pend all USB communications – both for the UMS and for the GWQ protocol. Although USB communication is reliable, the preamble and CRC-8 which bound the standard 8 byte header provide an easy frame synchronization method that provides both the host driver and the underlying hardware with an additional layer of data integrity protection.

The field names “chip_address” and “core_address” may not be relevant to many USB operations, but the names are retained for consistency with the data structure definition as presented in Chapter 1. In some cases, we define an equivalent structure, for convenience.

3.3.1 OP_USB_INIT – Initialize USB operations

This operation initializes the device, to prepare it for subsequent hash operations. Although there are various options, the command in its simplest form represents a one stop “set and go” option for activating the device. This operation includes powering up the device, and running startup diagnostics. Since the USB interface is active even on standby power, users who deploy systems consisting of large numbers of devices can power equipment on and off from a central control point, or even remotely.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_USB_INIT
chip_address	Reserved. Must be 0.
core_address	Protocol options – see below
hdata	Initialization options and power control – see below
data_length	Variable – see below

core_address field	Operation, and meaning of core_address field
bits 2:0	Protocol to use: 0 = USB mapped serial (UMS) protocol 1 = Global Work Queue (GWQ) protocol 2-7 = Reserved
3	Set to override configuration data with user supplied configuration in data block[13]
bits 7:4	Reserved. Must be 0.

13 Care must be taken, when using the GWQ protocol, that the options supplied here do not interfere with the underlying operation of the protocol. Users who wish to override configuration options should do so by first issuing an OP_USB_INIT without configuration options to acquire the default values in use, and then perform a second OP_USB_INIT with only configuration options that the user requires modified, the remainder unchanged.

hdata field	Operation, and meaning of hdata field
bits 11:0	Desired hash clock frequency Minimum value allowed: 25% greater than reference clock frequency Maximum value allowed: See below
12	If set, forces PLL bypass, so PLL is inactive, hash clock = reference clock
13	Set to disable automatic ASIC initialization sequence, host will perform this manually
14	Set to perform at speed individual core tests, return results in a second bitmap
15	Set to initialize the USB link only, but leave all underlying systems powered down

A convenience header is provided for the host to device direction, as follows:

```
struct hf_usb_init_header {
        uint8_t  preamble;                      // Always 0xaa
        uint8_t  operation_code;
        uint8_t  spare1

        uint8_t  protocol:3
        uint8_t  user_configuration:1;
        uint8_t  spare2;

        uint16_t hash_clock:12;                 // Requested hash clock frequency
        uint16_t pll_bypass:1;                  // Force PLL bypass, hash clock = ref clock
        uint16_t no_asic_initialization:1;      // Do not perform automatic ASIC initialization
        uint16_t do_atspeed_core_tests:1;       // Do core tests at speed, return second bitmap
        uint16_t leave_powered_down:1;          // Init USB only, leave device powered down

        uint8_t  data_length;                   // .. of data frame to follow, in 4 byte blocks
        uint8_t  crc8;                          // Computed across bytes 1-6 inclusive
        } __attribute__((packed,aligned(4)));   // 8 bytes total
```

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_USB_INIT
chip_address	Number of devices present in the underlying ASIC chain.
core_address	# of cores in each die on each ASIC
hdata	Device ID in b7:0, Reference clock rate (Mhz) in b15:8
data_length	Data length, variable – see below

Several useful items are returned in the *hdata* field. The LS byte contains the ASIC revision number. The MS byte contain the reference clock value, in MHz. Values of device ID are:

Device ID	Description
0	Reserved
1	Hashfast GN ASIC, Reference clock rate = 125 Ghz
128	FPGA Emulator. Hash clock rate = 250 Mhz (Development use only)

The first data returned after the header is a 16 byte structure as follows:

```
struct hf_usb_init_base {
        uint16_t firmware_rev;                  // Firmware revision #
        uint16_t hardware_rev;                  // Hardware revision #
        uint32_t serial_number;                 // Board serial number
        uint8_t  operation_status;              // Reply status for OP_USB_INIT
        uint8_t  extra_status_1;                //
        uint8_t  extra_status_2;                //
        uint8_t  extra_status_3;                //
        uint16_t hash_clockrate;                // Actual hash clock rate (nearest Mhz)
        uint16_t inflight_target;               // Target inflight amount for GWQ protocol
        } __attribute__((packed,aligned(4)));
```

The next segment of data returned is a structure corresponding to all of the settings used internally for an OP_CONFIG operation (see section 2.4.2.6 on page 35), described in the **struct hf_config_data** data structure definition. This is always returned, giving the user the default settings that firmware release is using. The user may use this structure, with certain fields modified, in a subsequent OP_USB_INIT operation, or directly in an OP_CONFIG operation; however, if the Global Work Queue protocol is being used, care must be taken not to modify fields which affect operation of the protocol. Contact HashFast support if further information on this matter is desired.

The third, variable length, data field is a bitmap giving a good/bad indication of core status. This field is not returned if the user disabled automatic initialization of the ASIC subsystem (bit 13 of the hdata field). Assuming it is returned, it will be of length (in bits) corresponding to the total number of cores in the system, rounded up to the next highest 32 bits.

Example:

> From the returned packet header, we note there are 96 cores per die, and 12 die total. Therefore, there are 1,152 cores in the system. This corresponds to 36 bytes, which is an even multiple of 4, so no rounding up is required.
>
> The core bitmap is ordered little endian, in die order. A bit set means the core is good. A bit clear means the core did not pass diagnostics and should not be used. Cores that are unuse-able are generally so due to semiconductor manufacturing yield. *HashFast products are over-specified and meet the published specifications at nominal clock rates even when there are several non-functional cores.*

Q. *A user notes that all 36 bytes of the returned data have all bits set, except for the third byte, which contains the value 0xffef. Which core is faulty?*

A. *Core #20 of Die #0, where the core numbering starts at 0.*

Users who select the Global Work Queue (GWQ) protocol do not need to interpret the returned core diagnostic map. The underlying GWQ protocol will already have taken into account any non-functional cores and no work will be queued to them. Users who use the mapped serial protocol will need to note any non-functional cores, and not queue work to them.

Finally, if the user specified the option to do at-speed individual core tests, bit 14 of the hdata field, then a second bitmap will be returned, giving the status of individual core tests. The format for the bitmap is identical to the format for the first bitmap.

Users who elect to over-clock may wish to use this option in order to detect cores that will not perform at the selected clock speed. However, since these tests are done on an individual basis, they may not accurately depict operation of a given core when all other cores on the device are hashing, since switching noise will be higher in the latter case. Users who over-clock should plan to introduce occasional test cases to each core, and/or to monitor nonce return statistics, to ensure that they are not in fact compromising performance by over-clocking at too high a rate.

Note: HashFast does not limit over-clocking; however, users who elect to over-clock must not override system thermal protections, or otherwise operate the equipment in a dangerous manner. Over-clocking is the user's sole responsibility; such operation will be recorded by the device in non-volatile memory, and will void equipment warranty.

3.3.1.1 OP_USB_INIT - Additional options

As described, the OP_USB_INIT frame provides the host with ample capability to prepare the device for work in one single, short operation. It is possible for the host to provide a number of additional options, in an associated data frame, to override certain initialization defaults. The host simply appends the option data to the header as usual, with a non-zero data_length field in the header.

The options available include the following. We note in particular the comments included in the protocol header file:

```
// Options (only if present) that may be appended to the above header
// Each option involving a numerical value will only be in effect if the value is non-zero
// This allows the user to select only those options desired for modification. Do not
// use this facility unless you are an expert - loading inconsistent settings will not work.
struct  hf_usb_init_options {
        uint16_t group_ntime_roll;                  // Total ntime roll amount per group
        uint16_t core_ntime_roll;                   // Total core ntime roll amount
        uint8_t  low_operating_temp_limit;          // Lowest normal operating limit
        uint8_t  high_operating_temp_limit;         // Highest normal operating limit
        uint16_t spare;
        } __attribute__((packed,aligned(4)));
```

3.3.2 OP_USB_SHUTDOWN – Shutdown operations

This short packet is a one-stop shutdown mechanism. Regardless of the status of the device, and what hashing operations are in progress, this command will shut down all activity and place the device in power-off standby mode. The user may elect to not power-off

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_USB_SHUTDOWN
chip_address	Reserved. Must be set to 0.
core_address	Reserved. Must be set to 0.
hdata	Bit 0 = power control Bit 0 set = Do not power down Bit 0 clear = Power down normally after shut down. Bit 1 = response control Bit 1 set = Send a reply Bit 1 clear = Do not send a reply
data_length	Always 0

The host will only get a response back if it specifically asked for one, by setting bit 1 of the hdata field. If it did ask for one, the response will be a short packet as follows:

Host gets back:

Field	Value/meaning (host to device)
operation_code	OP_USB_SHUTDOWN
chip_address	Reserved
core_address	Reserved
hdata	Completion status – always 0
data_length	Always 0

The completion status is always zero – this response is purely to provide the host with an active token confirming that the device is shut down. After getting back the OP_USB_SHUTDOWN response, the host will receive no further communications from the device until an OP_USB_INIT is performed.

3.3.3 OP_DIE_STATUS – On die sensors and related measurements

Periodically, the USB device will sent OP_DIE_STATUS blocks, containing one or more (usually two) blocks of sensor data from individual die, and the related board level power supply. These measurements include:

- All of the on-die sensor measurements, as described in the hf_g1_monitor structure definition in section 2.4.2.4. This includes die temperature measurement, and up to 6 core voltage measurement points.
- Temperature measurement of the board area around the power supply components associated with this die.
- Voltage and Current measurements at the board level for the power supply associated with this die, if supported by the board revision.[14]
- A fan or motor tachometer measurement, if available.

Across the four die on in individual GN ASIC, this amounts to 56 sensors per ASIC/module assembly. As noted in a footnote, this number will most likely diminish in later board revisions, in which case the corresponding board level sensor values will be reported as 0. Drivers should specifically allow for this outcome in the case of board level phase currents and voltage.

There are 4 fan connectors per module assembly. One tachometer reading is included in each OP_DIE_STATUS structure, so that across the four die on an ASIC the four fan tachometers are covered. These are, in die address order:

1. Waterpump tachometer. The waterpump always runs “flat out”
2. Radiator push (if fitted)
3. Radiator pull (if fitted)
4. Chassis fan (if fitted)

Which fan(s) are fitted depend on the box or chassis level design. There is always a waterpump and at least one radiator fan.

```
struct hf_g1_die_data {
        struct hf_g1_monitor die;                   // Die sensors - 8 bytes
        uint16_t phase_currents[4];                 // Phase currents (0 if unavailable)
        uint16_t voltage;                           // Voltage at device boundary (0 if unavailable)
        uint16_t temperature;                       // Regulator temp sensor
        uint16_t tacho;                             // See documentation
        uint16_t spare;
        } __attribute__((packed,aligned(4)));       // 24 bytes total
```

14 Initial production boards are over-engineered with regard to sensors and some measurement points may be removed from later revisions of the module.

The reporting interval for OP_DIE_STATUS frames is identical to (and in fact determined by) the periodic reporting interval for the underlying OP_STATUS frames which provide the on-chip sensor data. This interval is specified at startup time in the underlying OP_CONFIG operation, as described in section 2.4.2.6 on page 35. The relevant field is the status_period field, which defaults to reporting every 500 msec. This operation is transparent to USB users, who issue the OP_USB_INIT frame as a single do-it-all system startup procedure. However, the user does have the option of specifying the underlying configuration data, or reading the default values back from an initial OP_USB_INIT frame before specifying different parameters as an over-ride.

Conversion equations for the on-die sensor data are as given in section 2.4.2.6 on page 35.

Conversion equations for the on-board sensors are as follows:

- Voltage: Voltage (volts) = 16 bit integer reading * 19.0734e10-6 (full scale is 1.25 Volts)
- Phase current (amps) = 16 bit integer reading * 0.794728597e-3 (full scale is 52.083 Amps)
- Board temperature. *Conversion equation will be published in a later revision of this manual.*
- Tachometer: Direct tachometer reading in RPM.

To derive the total die current consumption, add the four phase currents.

OP_DIE_STATUS frames are always sent, regardless of whether the UMS or GWQ protocol is in use by the host.

3.3.4 OP_HASH operations over USB

OP_HASH operations form the fundamental way of getting a work block from the host to the device, regardless of whether the UMS or GWQ protocol are being used. A slightly modified data structure is used with OP_HASH operations across USB so that the combination of 8 byte header + OP_HASH data fits inside 64 bytes. The modified data structure is:

```
struct hf_hash_usb {
        uint8_t  midstate[32];                  // Computed from first half of block header
        uint8_t  merkle_residual[4];            // From block header
        uint32_t timestamp;                     // From block header
        uint32_t bits;                          // Actual difficulty target for block header
        uint32_t starting_nonce;                // Usually set to 0
        uint32_t nonce_loops;                   // How many nonces to search, or 0 for 2^32
        uint16_t ntime_loops:12;                // How many times to roll timestamp, or 0 [15]
        uint16_t spare1:4;
        uint8_t  search_difficulty;             // Search difficulty to use, # of '0' digits required
        uint8_t  group;                         // Non-zero for valid group
        } __attribute__((packed,aligned(4)));   //
```

The only difference between this structure and that used over the ASIC serial link is the dropping of the option byte, and removal of the crc32. The group code by convention is valid if it is non-zero, hence the option byte is not required.

Bulk OP_HASH operations may be dispatched by concatenating up to 8 at a time and dispatching them in a single USB write operation, this operation will usually be carried atomically across a 512 byte USB 2.0 buffer.

15 Always set this field to 0 for the G-1 stepping.

3.3.5 Directly mapped operations

The following operation types are directly mapped from the UMS protocol to the lower level ASIC serial protocol (firmware adds the necessary CRC-32 when required, and strips it on returned frames):

- OP_ROOT
- OP_UNROOT
- OP_ABORT
- OP_GPIO
- OP_GROUP
- OP_CLOCKGATE

and their corresponding descriptions in Chapter 2 may be used directly.

3.4 USB Global Work Queue (GWQ) protocol

> HashFast recommends that all drivers are written against this protocol. It is by far the simplest method for the host to control mining appliances, with dramatically reduced work traffic, complexity and CPU utilization in the host.

This protocol may be used in lieu of the mapped serial protocol, to significantly reduce the workload required within the mining client software. The communication model is reduced to the simplest possible case – a queue of WORK flows from the host to the device, and a queue of NONCES flows from the device back to the host, as well as advice on WORK COMPLETION.

Work generation occurs in the host as usual. Optionally, the device may self generate derivative work from each given hash job, by performing ntime rolling across a short range of time. This range is usually set to 24 seconds, reducing the hash job traffic volume by a factor of 24. This small range can easily be accommodated by host software when using the Stratum protocol to communicate with mining pools.

The sequence number space required to keep track of work is collapsed down to a small range if ntime rolling is allowed. The details of how work derivation is performed is of no concern to the host, except to note that the "ntime offset" field in returned nonces needs to be accounted for when submitting work to the pool server. Since the coinbase is periodically updated, and moreover completely replaced whenever a block is found on the network, a mechanism is defined to perform a work restart, to quickly flush old work out of hash cores and replace it with fresh work. Flushing stale work is an easy one step operation for the host when using the GWQ protocol.

To initialize the GWQ protocol, the host must issue an OP_USB_INIT exchange, with

- GWQ protocol (1) specified in b2:0 of the core_address field
- Do not disable internal system configuration

In this case, the returned OP_USB_INIT packet tells the host how the device is set up

- The *inflight_target* field of the returned base initialization structure defines how many jobs the host should try to maintain.
- The *sequence_modulus* field of the returned base initialization structure tells the host how it needs to wrap sequence numbers. For example, if the returned *sequence_modulus* is 1024, the host would wrap sequence numbers from 1023 back to 0.

The host may immediately queue exactly this number of OP_HASH operations. The first OP_HASH should start with sequence #1. Successive OP_HASH packets must be numbered sequentially, wrapping from *(sequence_modulus – 1)* back to 0.

OP_GWQ_STATUS packets will periodically be returned to the host, giving overall status of the device and indicating how much more work, if any, can be queued to the device. Returned sequence

numbering of the most recently completed work allows the host to free locally owned work resources as they age. See section 3.6.5 on page63 for an example of sequence number progression.

Most importantly, OP_NONCE packets will also be returned to the host, containing detected nonces. The host matches returned nonces with in-flight work using the sequence number returned along with each nonce, just as it does in the serial protocols.

Finally, OP_DIE_STATUS packets will periodically be sent to the host, giving detailed operating information for each module including all temperatures, voltages, fan speeds etc.

3.4.1 OP_GWQ_STATUS notifications

There are three sequence numbers of relevance to the host:

A) The most recently sent sequence number

B) The most recent sequence number the device knows about

C) The most recent sequence number of a completed job in the device

The difference between (A) and (B) represents the jobs that are "in flight" to the device. The difference between (B) and (C) are the number of active jobs within the device at the time of reporting. The difference between (A) and (C) therefore represents the number of jobs that are either in flight to the device or active in the device.

The OP_GWQ_STATUS packet contains the sequence numbers for (B) and (C) above. The host knows what (A) is. Therefore, all the host driver needs to do is a simple calculation to determine if it can send more work. Although (B) is provided by the device, it is not required.

For the GWQ protocol, the amount of work the device can absorb is dependent upon how much ntime rolling is configured into the device, which directly determines how many work items the device internally derives from each work item provided by the host. The host need not be concerned about these internal mechanisms, and to help the host determine how much work should be in flight, the OP_USB_INIT base information block provides an exact number – the "inflight_target" field as described in the hf_usb_init_base data structure in section 3.3.1.

The calculation of how much more work the host needs to send then is:

$$space\ for\ more\ jobs = inflight\_target - ((A) - (C))$$

where:

inflight_target is provided by the device in the reply to the OP_USB_INIT packet

(A) is the most recent sequence number sent from the host to the device

(c) is the sequence number tail sent from the device back to the host

If there is space for jobs, the host queues the calculated number of OP_HASH packets to the device. If the host gets the above calculation wrong, and sends more jobs, they will be discarded. However, the

sequence number mechanism will eventually catch up and skip over the lost jobs, allowing the host to recover them.

A slight addition to the above calculation occurs in practice. The device may load shed for thermal control (especially if the user attempts to over-clock excessively), in which case cores are shut down. A "shed count" field is returned in OP_GWQ_STATUS packets, and this amount must be subtracted from the result in order to determine the true job space. The sample driver code later on in this chapter illustrates the precise calculation. The "shed count" is in fact twice the number of cores shut down.

The OP_GWQ_STATUS notification also returns an unsigned 64 bit integer, which is the number of hashes completed since the last OP_GWQ_STATUS message. The host can add this to a cumulative counter, to keep track of total running hash rates.

OP_GWQ_STATUS messages are sent to the host periodically, typically every quarter to half second. There is no need for them to be faster because work is double buffered in every core, and cores typically take several seconds or more to complete a full nonce range hash search. An exception to the notification period is when an OP_WORK_RESTART is sent by the host, in which case an OP_GWQ_STATUS will be queued back to the host in reply so that "new" work can be issued immediately.

```
struct hf_gwq_data {
        uint64_t hash_count;                    // Add this to host's cumulative hash count
        uint16_t sequence_head;                 // The latest, internal, active sequence #
        uint16_t sequence_tail;                 // The latest, internal, completed sequence #
        uint16_t shed_count;                    // # cores have been shed for thermal control
        uint16_t spare;
        } __attribute__((packed,aligned(4)));
```

3.4.2 OP_WORK_RESTART (GWQ protocol only)

The Stratum protocol periodically refreshes the set of transactions that mining applications create work from, typically every 30 seconds. Since the host does not deal with core management when using the GWQ protocol, a fast, seamless mechanism is made available to abort all in-progress work and replace it with new work that reflects the new coinbase.

To perform a work restart, the host driver sends an OP_WORK_RESTART packet. The device will immediately abort all current work, and also immediately send back an OP_GWQ_STATUS packet that will (i) advance the tail pointer up to the head pointer, indicating that all work is done, and (ii) contain a hash count field which represents the extent of the work aborted, but not yet reported.

Upon seeing the tail pointer change, the host driver's normal mechanisms will free all of the old work, and issue new work. No special procedures need be followed by the host – the changeover is seamless and can typically be achieved in under a msec.

Host sends:

Field	Value/meaning (host to device)
operation_code	OP_WORK_RESTART
chip_address	Set to HF_GWQ_ADDRESS
core_address	Reserved. Must be set to 0.
hdata	Reserved. Must be zero.
data_length	Always 0

Special note:

HashFast expects to improve this feature before product release, such that it will not be necessary for the host to wait for a returned OP_GWQ_STATUS before sending the replacement work. We expect to allow the host to be able to queue at least a complete set of new work (inflight_target # of OP_HASH operations / 2), with the understanding that the work will always be accepted immediately after processing of the OP_WORK_RESTART operation. We can do this because of the flow through nature of operations in the underlying ASIC serial interface, and the fact that operations such as OP_RESET and OP_HASH get performed in real time as the associated serial packet flows through the daisy chained ASICs.

Driver developers should monitor the HashFast support pages for supplemental information, or an update to this manual, confirming the availability of this feature.

3.4.3 Thermal control with the GWQ protocol

Mining applications today often perform thermal control of mining equipment, operating equipment between a pair of temperature control limits by adding or removing work from hash cores. In the case of the GWQ protocol, the mining application does not have the degree of control required to do this at a granular enough level. Aborting work or starving the device of work in the GWQ protocol would shed many cores at once, due to the work multiplication methods inherent to the GWQ protocol.

HashFast products perform their own thermal control when the GWQ protocol is used. Default temperature limits are employed, and individual cores are shut down and restarted in accordance with a control system designed to keep the equipment within these limits. These actions are easy to perform internally with the GWQ protocol – we simply shut down or re-activate cores within groups. This is not completely transparent to the host, because

1. Hash rates will drop during the periods that cores are shut down, and
2. A “shed_count” is returned several times per second to the host noting how many work items, if any, will not be used due to inactive cores. This count is included in the information provided with the OP_GWQ_STATUS message.

The host may modify the default temperature limits by optionally providing override limits at the time the device is initialized. However, these limits must be sensible, and limits that are not sensible (high lower than low, high higher than thermal shutdown limits etc.) will be rejected by the firmware.

3.5 USB statistics collection

Several types of statistics frames may be periodically sent to the host. The host driver may choose to ignore these frames, or accumulate the statistics contained therein and make those statistics available to the user.

3.5.1 OP_USB_STATS1

This information frame periodically returns communication statistics. Host drivers should accrue the statistics locally, by adding the returned values to (wider) locally held counters. These statistics are rarely of much interest to anyone except a system administrator.

```
struct hf_usb_stats1 {
        // USB incoming
        uint16_t usb_rx_preambles;
        uint16_t usb_rx_receive_byte_errors;
        uint16_t usb_rx_bad_hcrc;

        // USB outgoing
        uint16_t usb_tx_attempts;
        uint16_t usb_tx_packets;
        uint16_t usb_tx_timeouts;
        uint16_t usb_tx_incompletes;
        uint16_t usb_tx_endpointstalled;
        uint16_t usb_tx_disconnected;
        uint16_t usb_tx_suspended;

        // Internal UART transmit
        uint16_t uart_tx_queue_dma;
        uint16_t uart_tx_interrupts;

        // Internal UART receive
        uint16_t uart_rx_preamble_ints;
        uint16_t uart_rx_missed_preamble_ints;
        uint16_t uart_rx_header_done;
        uint16_t uart_rx_data_done;
        uint16_t uart_rx_bad_hcrc;
        uint16_t uart_rx_bad_dma;
        uint16_t uart_rx_short_dma;
        uint16_t uart_rx_buffers_full;

        uint8_t  max_tx_buffers;                    // Maximum # of send buffers ever used
        uint8_t  max_rx_buffers;                    // Maximum # of receive buffers ever used
        } __attribute__((packed,aligned(4)));
```

3.5.2 OP_USB_STATS2

[Will be provided in a future revision]

3.6 Sample driver code fragments

We provide here relevant GWQ driver code fragments, extracted from a sample CGMiner driver. The code has been simplified – removal of locking etc. - to make it more readable. The following fragments are not the complete driver, just the main initialization, write and read threads, along with some other illustrative fragments.

3.6.1 Device initialization

The following sample device initialization code makes extensive use of the structure definitions provided by HashFast. Device initialization may occur from a powered down state. The USB interface uses standby power, so it is possible for a host driver to power-up an entire rack of equipment. The code below notes that it can take over a second to power-up equipment, and as such the normal receive response timeout is extended.

The steps involved in the following initialization code are:

1. Assemble a simple OP_USB_INIT packet, and send it.
2. Loop waiting for a valid reply. Once a valid reply is received, save the various information fields and initialize the host's sequence numbers.
3. Read and save the base data (struct usb_init_base – always returned). Note that this base data contains the important *inflight_target* field – this will be used later in the write thread.
4. Read and save the configuration data, and the core map.
5. The system is now ready. Separate code would then start up the write and read threads.

```
static int hashfast_reset(struct cgpu_info *hashfast)
    {
    hf_info_t *info = hashfast_infos[hashfast->device_id];
    struct hf_usb_init_header usb_init, *hu = &usb_init;
    struct hf_usb_init_base *db;
    uint8_t buf[256] __attribute__((packed,aligned(4)));
    struct hf_header *h = (struct hf_header *)buf;
    uint8_t hcrc;
    int i, ret;
    int response_retries;

    // Assemble the USB_INIT request packet
    memset(hu, 0, sizeof(*hu));
    hu->preamble = HF_PREAMBLE;
    hu->operation_code = OP_USB_INIT;
    hu->protocol = PROTOCOL_GLOBAL_WORK_QUEUE;
    hu->hash_clock = DEFAULT_HASH_CLOCK_DATE;
    hu->crc8 = hf_crc8((uint8_t *)hu);

    for (i = 0; i < RESET_TRIES; i++)
        {
        // Flush any old buffered input or output
        usb_buffer_clear(hashfast);

        // Send the OP_USB_INIT frame
        if (hashfast_send_header(hashfast, (struct hf_header *)hu))
            {
            cgsleep_ms(200);
            continue;
            }

        // Check for the correct response.
        // We extend the normal timeout - a complete device initialization, including
        // bringing power supplies up from standby, etc., can take over a second.
        for (response_retries = 0; response_retries < 20; response_retries++)
```

```
        if ((ret = hashfast_get_header(hashfast, h, &hcrc)) != 0)
            break;
    // The following checks are overkill, but we play it safe and do them all
    if (ret == 0)
        applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! Reply timeout", hashfast->device_id);
    else if (ret < 0)
        applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! Receive error core %d", hashfast->device_id, ret);
    else if (ret != sizeof(*h)
     || (h->preamble != HF_PREAMBLE)
     || (h->crc8 != hcrc))
        {
        applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! ret %d, hcrc 0x%02x, Unknown reply:", hashfast->device_id, ret, hcrc);
        continue;
        }
    else if (h->operation_code != OP_USB_INIT)
        {
        applog(LOG_WARNING, "HF%d: OP_USB_INIT: Tossing packet, valid but unexpected type", hashfast->device_id);
        if (h->data_length)
            hashfast_get_data(hashfast, (uint8_t *) buf, h->data_length);
        continue;
        }
    else
        {
        // Here for a good reply
        applog(LOG_DEBUG, "HF%d: OP_USB_INIT: %d die in chain, %d cores, device_type %d, refclk %d Mhz",
                hashfast->device_id, h->chip_address, h->core_address, h->hdata&0xff, (h->hdata>>8)&0xff);

        info->asic_count = h->chip_address;             // Save device configuration
        info->core_count = h->core_address;
        info->device_type = (uint8_t)h->hdata;
        info->ref_frequency = (uint8_t)(h->hdata>>8);
        info->hash_sequence_head = 0;                   // Initialize local sequence #'s
        info->hash_sequence_tail = 0;
        info->device_sequence_tail = 0;
        info->shed_count = 0;

        // Size in bytes of the core bitmap
        info->core_bitmap_size = (((info->asic_count * info->core_count) + 31) / 32) * 4;

        // Get the usb_init_base structure, print out a few useful fields
        if ((hashfast_get_data(hashfast, &info->usb_init_base, U32SIZE(info->usb_init_base))) != U32SIZE(info->usb_init_base))
            {
            applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! Failure to get usb_init_base data", hashfast->device_id);
            continue;
            }
        db = &info->usb_init_base;
        applog(LOG_DEBUG, "HF%d:      firmware_rev:    %d.%d", hashfast->device_id, (db->firmware_rev>>8)&0xff,db->firmware_rev&0xff);
        applog(LOG_DEBUG, "HF%d:      hardware_rev:    %d.%d", hashfast->device_id, (db->hardware_rev>>8)&0xff,db->hardware_rev&0xff);
        applog(LOG_DEBUG, "HF%d:      serial number:   %d", hashfast->device_id, db->serial_number);
        applog(LOG_DEBUG, "HF%d:      hash clockrate:  %d Mhz", hashfast->device_id, db->hash_clockrate);
        applog(LOG_DEBUG, "HF%d:      inflight_target: %d", hashfast->device_id, db->inflight_target);
        applog(LOG_DEBUG, "HF%d:      sequence_modulus: %d", hashfast->device_id, db->sequence_modulus);

        // Now get and save the config data used.
        if ((hashfast_get_data(hashfast, (uint8_t *) &info->config_data, U32SIZE(info->config_data))) != U32SIZE(info->config_data))
            {
            applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! Failure to get config_data", hashfast->device_id);
            continue;
            }

        // Now the core bitmap
        if (!(info->core_bitmap = malloc(info->core_bitmap_size)))
            quit(1, "Failed to malloc core_bitmap");
        if ((hashfast_get_data(hashfast, (uint8_t *)info->core_bitmap, info->core_bitmap_size/4)) != info->core_bitmap_size/4)
            {
            applog(LOG_WARNING, "HF%d: OP_USB_INIT failed! Failure to get core_bitmap", hashfast->device_id);
            continue;
            }
        applog(LOG_DEBUG, "HF%d: OP_USB_INIT: Core bitmap returned:", hashfast->device_id);
        hexdump((uint8_t *)info->core_bitmap, info->core_bitmap_size);

        // That's it, we're all initialized
        return(0);
        }
    }

// Only here if initialization failed
applog(LOG_ERR, "HF%d: Giving up", hashfast->device_id);
return 1;
}
```

3.6.2 Write thread

The only purpose of the write thread is to deliver work to the device. There are three simple steps in the write thread, and we loop forever (until shutdown) performing them. These steps are:

1. If work is to be restarted, we send an OP_WORK_RESTART frame. That's all we need to do, the hardware will take care of everything required.
2. We compute a *jobs_to_send* variable. We use the *inflight_target* field from the previously saved initialization data, the latest sequence number we sent (*info->hash_sequence*), and the tail sequence number from the device (*info->device_sequence_tail*) previously saved by the read thread (see next section). We use the SEQUENCE_DISTANCE() macro provided in the HashFast header file to compute the circular difference. If the result is non zero, we can send work.
3. To send work, we get a work item from the mining application, and fill in an OP_HASH data block. Note how we compute *search_difficulty*. work->device_diff is a (double) difficulty level to apply to the device, >= 1.0. We truncate this to an integer, and keep halving it while incrementing a variable, finally reaching the quantity we desire – the number of leading 0's that qualify nonce candidates for return to the host. We use the next sequence number, modulo 256. After queuing the OP_HASH frame to the device, we remember the sequence number we just sent, and the "work" that corresponds to this sequence number.

```
////////////////////////////////////////////////////////////////////////////
// Global Work Queue protocol - write thread
////////////////////////////////////////////////////////////////////////////
static void *hf_gwq_write(void *stuff)
  {
  struct thr_info *thr = stuff;
  struct cgpu_info *hashfast = (struct cgpu_info *) thr->cgpu;
  hf_info_t *info = hashfast_infos[hashfast->device_id];
  struct hf_hash_usb op_hash_data;
  struct work *work;
  uint64_t intdiff;
  uint32_t *p;
  int16_t jobs_to_send;
  int i, sequence;

  memset(&op_hash_data, 0, sizeof(op_hash_data));
  while (likely(!hashfast->shutdown))
    {
    if (info->work_restart)
      {
      // Stratum protocol has commenced a new job, so what we are working on right now has become stale.
      // Queue an abort to the device. We use the USB specific OP_WORK_RESTART function, which methodically
      // runs down all active hashing in the device. The device sequence tail will fast-forward, allowing
      // us to quickly and seamlessly replace the run down work.
      info->work_restart = false;
      hashfast_send_frame(hashfast, OP_WORK_RESTART, HF_GWQ_ADDRESS, 0, 0, (uint8_t *)NULL, 0);
      }

    // Figure out how many jobs we can send. The device gave us the "inflight target" during initialization.
    jobs_to_send = info->usb_init_base.inflight_target - info->shed_count -
                                          SEQUENCE_DISTANCE(info->hash_sequence_head,info->device_sequence_tail);
    if (likely(jobs_to_send <= 0))    // Can be < 0 if device shed cores for thermal control
      {
      cgsleep_ms(10);         // Nothing to do
      continue;
      }

    while (jobs_to_send > 0)
      {
      work = get_work(thr, thr->id);
      if (unlikely(thr->work_restart))
        {
        free_work(work);
        break;
        }
```

```
    // Assemble the data frame and send the OP_HASH packet
    memcpy(op_hash_data.midstate, work->midstate, sizeof(op_hash_data.midstate));
    memcpy(op_hash_data.merkle_residual, work->data + 64, 4);
    p = (uint32_t *) (work->data + 64 + 4);
    op_hash_data.timestamp = *p++;
    op_hash_data.bits = *p++;
    op_hash_data.nonce_loops = info->hash_loops;

    // Set the number of leading zero's to look for based on difficulty
    intdiff = (uint64_t) work->work_difficulty;  // drv->max_diff is never a factor
    for (i = 31; intdiff; i++, intdiff >>= 1)   // Diff 1 = 32, Diff 2 = 33, Diff 4 = 34 etc.
      ;
    op_hash_data.search_difficulty = i;

    if ((sequence = info->hash_sequence_head+1) >= info->num_sequence)
      sequence = 0;

    if (hashfast_send_frame(hashfast, OP_HASH, HF_GWQ_ADDRESS, 0, sequence, (uint8_t *)&op_hash_data, sizeof(op_hash_data)))
      {
      applog(LOG_ERR, "HF%d: Error sending OP_HASH", hashfast->device_id);
      cgsleep_ms(1);
      break;
      }
    // Remember the work
    info->hash_sequence_head = sequence;
    jobs_to_send--;
    *(info->gwq_work + info->hash_sequence_head) = work;

    applog(LOG_DEBUG, "HF%d: OP_HASH (GWQ) sequence %d search_difficulty %d work 0x%x work_difficulty %g jobs_to_send %d",
      hashfast->device_id, info->hash_sequence_head, op_hash_data.search_difficulty, work, work->work_difficulty, jobs_to_send);
    }
  }
return NULL;
}
```

3.6.3 Read thread

The read thread processes periodically returned status information, and nonces. The *hashfast_get_packet()* function returns a validated packet. We use a switch statement and dispatch the operation, based on the packet header operation code *h->operation_code.*The cases handled in the code below are as follows:

1. OP_GWQ_STATUS frames. We simply update our local saved copies of the device sequence numbers (we only actually need the tail), and update our hash_count statistics with the returned value, which represents total hashes performed since the last OP_GWQ_STATUS frame. This then provides the *info->device_sequence_tail* used in the previously described write thread.

   If the sequence tail has progressed since the last OP_GWQ_STATUS we received, work has been completed, so we walk through each intermediate sequence number until our local copy of the tail pointer corresponds to the tail pointer we just received from the device, free'ing up the local work structures in the process.

2. OP_NONCE frames. This is what it's all about! Note how in this and other sections we use the structure definitions provided in the HashFast supplied header file – making the code readable and less prone to errors. For each returned nonce, we use the returned sequence number to find the corresponding work, and submit it. The GWQ protocol can roll ntime by a few seconds, this is taken into account by passing the ntime offset for the returned nonce into the *submit_offset_nonce()* function call provided by CGMiner.

   Very occasionally, the *n->search* field may be set, in which case the hardware is telling us that there is at least one more nonce in the next 128 successive nonce values. If so, we go and search for it/them, with a very brief throwback to CPU mining, via the function call shown.

3. OP_DIE_STATUS frames. We simply call a function to process the returned sensor data.

```
////////////////////////////////////////////////////////////////////////////
// Global Work Queue protocol - read thread
////////////////////////////////////////////////////////////////////////////
static void *hf_gwq_read(void *stuff)
  {
  struct cgpu_info *hashfast = (struct cgpu_info *) stuff;
  hf_info_t *info = hashfast_infos[hashfast->device_id];
  struct hf_candidate_nonce *n;
  struct hf_gwq_data *g;
  struct work *work;

  int num_nonces;
  int ret;
  int i;

  char buf[256];
  struct hf_header *h = (struct hf_header *)buf;

  // Loop forever reading and handling response packets
  while (likely(!hashfast->shutdown))
    {
    // Check for response and validate packet
    if ((ret = hashfast_get_packet(hashfast, h)) <= 0)
      continue;

    // Dispatch the operation
    switch (h->operation_code)
      {
      case OP_GWQ_STATUS:               // Global Work Queue status
        g = (struct hf_gwq_data *)(h+1);
        applog(LOG_DEBUG, "HF%d: GWQ: OP_GWQ_STATUS, device_head %4d tail %4d my tail %4d shed %3d inflight %4d", hashfast->device_id,
```

```
        g->sequence_head, g->sequence_tail, info->hash_sequence_tail, g->shed_count, SEQUENCE_DISTANCE(info->hash_sequence_head,g-
>sequence_tail));

       // Update local database
       info->hash_count += g->hash_count;
       info->device_sequence_head = g->sequence_head;
       info->device_sequence_tail = g->sequence_tail;
       info->shed_count = g->shed_count;

       // Free any work that is no longer required (actually operates one-behind)
       while (info->device_sequence_tail != info->hash_sequence_tail)
         {
         if (++info->hash_sequence_tail >= info->num_sequence)
           info->hash_sequence_tail = 0;
         if (unlikely(!(work = *(info->gwq_work + info->hash_sequence_tail))))
           quit(1, "Bad gwq_work sequence tail");
         applog(LOG_DEBUG, "HF%i: GWQ: Completing work 0x%x on hash_sequence_tail %d", hashfast->device_id, work, info->hash_sequence_tail);
         free_work(work);
         hashfast->queued--;
         *(info->gwq_work + info->hash_sequence_tail) = (struct work *)NULL;
         }
       break;

     case OP_NONCE:
       // One or more nonces have been returned. Process them.
       n = (struct hf_candidate_nonce *)(h+1);
       num_nonces = h->data_length / U32SIZE(sizeof(struct hf_candidate_nonce));
       applog(LOG_DEBUG, "HF%d: GWQ: OP_NONCE: %2d:, num_nonces %d hdata 0x%04x", hashfast->device_id, h->chip_address, num_nonces, h-
>hdata);
       for (i = 0; i < num_nonces; i++, n++)
         {
         applog(LOG_DEBUG, "HF%d: GWQ: OP_NONCE: %2d: %2d: search %1d ntime %2d sequence %4d nonce 0x%08x",
             hashfast->device_id, h->chip_address, i, n->search, n->ntime, n->sequence, n->nonce);
         // Find the job from the sequence number
         work = *(info->gwq_work + n->sequence);
         submit_noffset_nonce(info->thr, work, n->nonce, n->ntime);
         if (n->search)
           {
           applog(LOG_DEBUG, "HF%d: OP_NONCE: SEARCH PROXIMITY EVENT FOUND", hashfast->device_id);
           search_for_extra_nonce(hashfast, work, n);
           }
         }
       break;

     case OP_DIE_STATUS:
       update_die_status(hashfast, h);
       break;

     case OP_STATISTICS:
       update_die_statistics(hashfast, h);
       break;

     case OP_USB_STATS1:
       update_stats1(hashfast, h);
       break;

     default:
       break;
     }
   }
  return NULL;
  }
```

3.6.4 update_die_status() function

The update_die_status() function in the previously described read thread simply copies the returned sensor data into local thread storage. The data might typically also be copied into an API area (not shown below).

HashFast products perform their thermal monitoring, fan control and load shedding internally, so this data is for driver and API informational purposes only, no control action is required. In the code below we include a sample fragment that decodes the on-die sensor values, using the macro's provided for the purpose in the HashFast supplied header file.

```
////////////////////////////////////////////////////////////////////////////
// Update monitoring data from on-die sensors
// Board level power regulator voltage, current and temperature sensors of
// the supply for that die are sent in the same structure.
////////////////////////////////////////////////////////////////////////////
static void update_die_status(struct cgpu_info *hashfast, struct hf_header *h)
    {
    hf_info_t *info = hashfast_infos[hashfast->device_id];
    struct hf_g1_die_data *d = (struct hf_g1_die_data *)(h+1), *ds;
    int num_included = (h->data_length * 4) / sizeof(struct hf_g1_die_data);
    int i, j;

    // Copy in the data. They're numbered sequentially from the starting point
    ds = info->die_status + h->chip_address;
    for (i = 0; i < num_included; i++)
        memcpy(ds++, d++, sizeof(struct hf_g1_die_data));

#ifdef DEBUG
    // Example of converting returned values
    for (i = 0, d = &info->die_status[h->chip_address]; i < num_included; i++, d++)
        {
        float die_temperature;
        float core_voltage[6];

        die_temperature = G1_DIE_TEMPERATURE(d->die.die_temperature);
        for (j = 0; j < 6; j++)
            core_voltage[j] = G1_CORE_VOLTAGE(d->die.core_voltage[j]);

        applog(LOG_DEBUG, "HF%d: die %2d: OP_DIE_STATUS Die temp %.2fC vdd's %.2f %.2f %.2f %.2f %.2f %.2f",
                hashfast->device_id, h->chip_address + i, die_temperature,
                core_voltage[0], core_voltage[1], core_voltage[2], core_voltage[3], core_voltage[4], core_voltage[5]);
        }
#endif
    }
```

3.6.5 Sample driver trace – OP_GWQ_STATUS

The sample driver trace data below indicates how the sequence numbering scheme operates, in this case with a single 4 die ASIC, all default conditions. With 384 cores, the device returned an inflight target of twice this amount (768) and a sequence numbering modulus of 2048.

At startup the driver loads 768 work items into the device. While the 384 cores perform the first full nonce range hash (3+ seconds), the tail pointer does not move. Once these jobs complete, jobs that were pending in each core take over and the device tail pointer progresses, allowing the host to free work items. The progression of work through the device, with the device and local tail pointers trailing the head pointer progression, can clearly be seen. Moreover, the “inflight” count is always more than the number of cores - since all cores are always hashing - and the inflight number oscillates at or under the inflight target by tail pointer progression increments.

```
[2013-10-17 08:29:59] HF0: GWQ: OP_GWQ_STATUS, device_head    0 tail    0 my tail    0 shed   0 inflight    1
[2013-10-17 08:29:59] HF0: GWQ: OP_GWQ_STATUS, device_head   17 tail    0 my tail    0 shed   0 inflight   24
[2013-10-17 08:29:59] HF0: GWQ: OP_GWQ_STATUS, device_head   26 tail    0 my tail    0 shed   0 inflight   27
[2013-10-17 08:30:00] HF0: GWQ: OP_GWQ_STATUS, device_head  768 tail    0 my tail    0 shed   0 inflight  768
[2013-10-17 08:30:01] HF0: GWQ: OP_GWQ_STATUS, device_head  768 tail    0 my tail    0 shed   0 inflight  768
[2013-10-17 08:30:03] HF0: GWQ: OP_GWQ_STATUS, device_head  768 tail    0 my tail    0 shed   0 inflight  768
[2013-10-17 08:30:04] HF0: GWQ: OP_GWQ_STATUS, device_head  768 tail  192 my tail    0 shed   0 inflight  576
[2013-10-17 08:30:06] HF0: GWQ: OP_GWQ_STATUS, device_head  960 tail  384 my tail  192 shed   0 inflight  576
[2013-10-17 08:30:07] HF0: GWQ: OP_GWQ_STATUS, device_head 1152 tail  384 my tail  384 shed   0 inflight  768
[2013-10-17 08:30:07] HF0: GWQ: OP_GWQ_STATUS, device_head 1152 tail  417 my tail  384 shed   0 inflight  735
[2013-10-17 08:30:08] HF0: GWQ: OP_GWQ_STATUS, device_head 1185 tail  576 my tail  417 shed   0 inflight  609
[2013-10-17 08:30:08] HF0: GWQ: OP_GWQ_STATUS, device_head 1344 tail  586 my tail  576 shed   0 inflight  758
[2013-10-17 08:30:09] HF0: GWQ: OP_GWQ_STATUS, device_head 1354 tail  733 my tail  586 shed   0 inflight  621
[2013-10-17 08:30:09] HF0: GWQ: OP_GWQ_STATUS, device_head 1501 tail  768 my tail  733 shed   0 inflight  733
[2013-10-17 08:30:10] HF0: GWQ: OP_GWQ_STATUS, device_head 1536 tail  768 my tail  768 shed   0 inflight  768
[2013-10-17 08:30:10] HF0: GWQ: OP_GWQ_STATUS, device_head 1536 tail  768 my tail  768 shed   0 inflight  768
[2013-10-17 08:30:11] HF0: GWQ: OP_GWQ_STATUS, device_head 1536 tail  928 my tail  768 shed   0 inflight  608
[2013-10-17 08:30:12] HF0: GWQ: OP_GWQ_STATUS, device_head 1696 tail  960 my tail  928 shed   0 inflight  736
[2013-10-17 08:30:12] HF0: GWQ: OP_GWQ_STATUS, device_head 1728 tail 1096 my tail  960 shed   0 inflight  632
[2013-10-17 08:30:13] HF0: GWQ: OP_GWQ_STATUS, device_head 1864 tail 1152 my tail 1096 shed   0 inflight  712
[2013-10-17 08:30:13] HF0: GWQ: OP_GWQ_STATUS, device_head 1920 tail 1152 my tail 1152 shed   0 inflight  768
[2013-10-17 08:30:14] HF0: GWQ: OP_GWQ_STATUS, device_head 1920 tail 1152 my tail 1152 shed   0 inflight  768
[2013-10-17 08:30:14] HF0: GWQ: OP_GWQ_STATUS, device_head 1920 tail 1270 my tail 1152 shed   0 inflight  650
[2013-10-17 08:30:15] HF0: GWQ: OP_GWQ_STATUS, device_head 2038 tail 1344 my tail 1270 shed   0 inflight  694
[2013-10-17 08:30:16] HF0: GWQ: OP_GWQ_STATUS, device_head   64 tail 1536 my tail 1344 shed   0 inflight  576
[2013-10-17 08:30:17] HF0: GWQ: OP_GWQ_STATUS, device_head  256 tail 1536 my tail 1536 shed   0 inflight  768
[2013-10-17 08:30:17] HF0: GWQ: OP_GWQ_STATUS, device_head  256 tail 1536 my tail 1536 shed   0 inflight  768
[2013-10-17 08:30:18] HF0: GWQ: OP_GWQ_STATUS, device_head  256 tail 1728 my tail 1536 shed   0 inflight  576
[2013-10-17 08:30:19] HF0: GWQ: OP_GWQ_STATUS, device_head  448 tail 1780 my tail 1728 shed   0 inflight  716
[2013-10-17 08:30:20] HF0: GWQ: OP_GWQ_STATUS, device_head  500 tail 1920 my tail 1780 shed   0 inflight  628
[2013-10-17 08:30:20] HF0: GWQ: OP_GWQ_STATUS, device_head  640 tail 1920 my tail 1920 shed   0 inflight  768
[2013-10-17 08:30:21] HF0: GWQ: OP_GWQ_STATUS, device_head  640 tail 1920 my tail 1920 shed   0 inflight  768
[2013-10-17 08:30:21] HF0: GWQ: OP_GWQ_STATUS, device_head  640 tail 1953 my tail 1920 shed   0 inflight  735
[2013-10-17 08:30:22] HF0: GWQ: OP_GWQ_STATUS, device_head  673 tail   64 my tail 1953 shed   0 inflight  609
[2013-10-17 08:30:23] HF0: GWQ: OP_GWQ_STATUS, device_head  832 tail  242 my tail   64 shed   0 inflight  590
[2013-10-17 08:30:24] HF0: GWQ: OP_GWQ_STATUS, device_head 1010 tail  256 my tail  242 shed   0 inflight  754
[2013-10-17 08:30:25] HF0: GWQ: OP_GWQ_STATUS, device_head 1024 tail  256 my tail  256 shed   0 inflight  768
[2013-10-17 08:30:25] HF0: GWQ: OP_GWQ_STATUS, device_head 1024 tail  437 my tail  256 shed   0 inflight  587
[2013-10-17 08:30:26] HF0: GWQ: OP_GWQ_STATUS, device_head 1205 tail  448 my tail  437 shed   0 inflight  757
[2013-10-17 08:30:27] HF0: GWQ: OP_GWQ_STATUS, device_head 1216 tail  605 my tail  448 shed   0 inflight  611
[2013-10-17 08:30:28] HF0: GWQ: OP_GWQ_STATUS, device_head 1373 tail  640 my tail  605 shed   0 inflight  733
[2013-10-17 08:30:28] HF0: GWQ: OP_GWQ_STATUS, device_head 1408 tail  640 my tail  640 shed   0 inflight  768
[2013-10-17 08:30:29] HF0: GWQ: OP_GWQ_STATUS, device_head 1408 tail  832 my tail  640 shed   0 inflight  576
[2013-10-17 08:30:30] HF0: GWQ: OP_GWQ_STATUS, device_head 1600 tail  947 my tail  832 shed   0 inflight  653
[2013-10-17 08:30:31] HF0: GWQ: OP_GWQ_STATUS, device_head 1715 tail 1024 my tail  947 shed   0 inflight  691
[2013-10-17 08:30:31] HF0: GWQ: OP_GWQ_STATUS, device_head 1792 tail 1024 my tail 1024 shed   0 inflight  768
[2013-10-17 08:30:32] HF0: GWQ: OP_GWQ_STATUS, device_head 1792 tail 1024 my tail 1024 shed   0 inflight  768
[2013-10-17 08:30:32] HF0: GWQ: OP_GWQ_STATUS, device_head 1792 tail 1142 my tail 1024 shed   0 inflight  650
[2013-10-17 08:30:33] HF0: GWQ: OP_GWQ_STATUS, device_head 1910 tail 1216 my tail 1142 shed   0 inflight  694
[2013-10-17 08:30:33] HF0: GWQ: OP_GWQ_STATUS, device_head 1984 tail 1310 my tail 1216 shed   0 inflight  674
```

4 GN ASIC and HashFast product parameters

This Chapter is included to provide a brief outline of hardware parameters for HashFast products, or products from other vendors based on HashFast ASICs. It is provided as general background information for driver developers, and not any sort of hardware user guide.

The general parameters of a HashFast GN ASIC are:

- Each ASIC substrate contains 4 separate die.
- Each die contains 96 cores.
- Each core contains two complete double hash engines, which share work across one job
- Hash cores may nominally be clocked at 550 Mhz[16]
- Hash cores are rated for clocking at up to 700 Mhz under standard operating conditions
- Clock rates higher than this, and voltage levels lower than the nominal 0.81V, are outside of normal operating conditions.
- Cores search for nonces at (host specifiable) hardware based levels of Bitcoin Difficulty up to the ridiculously high limit of 7.922×10^{28}

Therefore, each GN "ASIC" looks like four addressable ASIC's in the context of the previous chapters in this document, providing a total of 96 * 4 * 2 = 768 double hash cores operating nominally at 550 Mhz, leading to a nominal hash rate of 422.4 GH/sec.

Operation at 700 Mhz leads to a total hash rate of 537.6 GH/sec, but sustained operation at this level may run into power distribution or thermal limitations,depending on cooling efficiency.

Operation beyond this clock rate, even if maintained within power and thermal limits, may lead to degraded hash performance as hash cores start to make mistakes. If attempting to do this, host software should monitor nonce rates and/or perform periodic testing of cores in order to set performance limits.

Modules which are over-clocked will contain logs of such operation, which may void warranty.

It is possible, due to yield, that some core(s) on some die may not work properly. A self test mechanism in the module micro-controller provides a way to advise the host so that these cores can remain unused. The guaranteed throughput of 400 GH/sec allows for a certain number of faulty cores.

16 Preliminary data – subject to change

4.1 HashFast GN module assembly

HashFast mining products employ a common board level assembly. Each 12” x 4” board contains one HashFast GN ASIC, a micro-controller, and power regulators for each die. Connectors exist for USB, fans, 12V power, and chaining connectors for linking several modules together.

The following deliberately vague renderings illustrate the module arrangement for interested users and driver developers, without giving too much away at this time to our competitors!

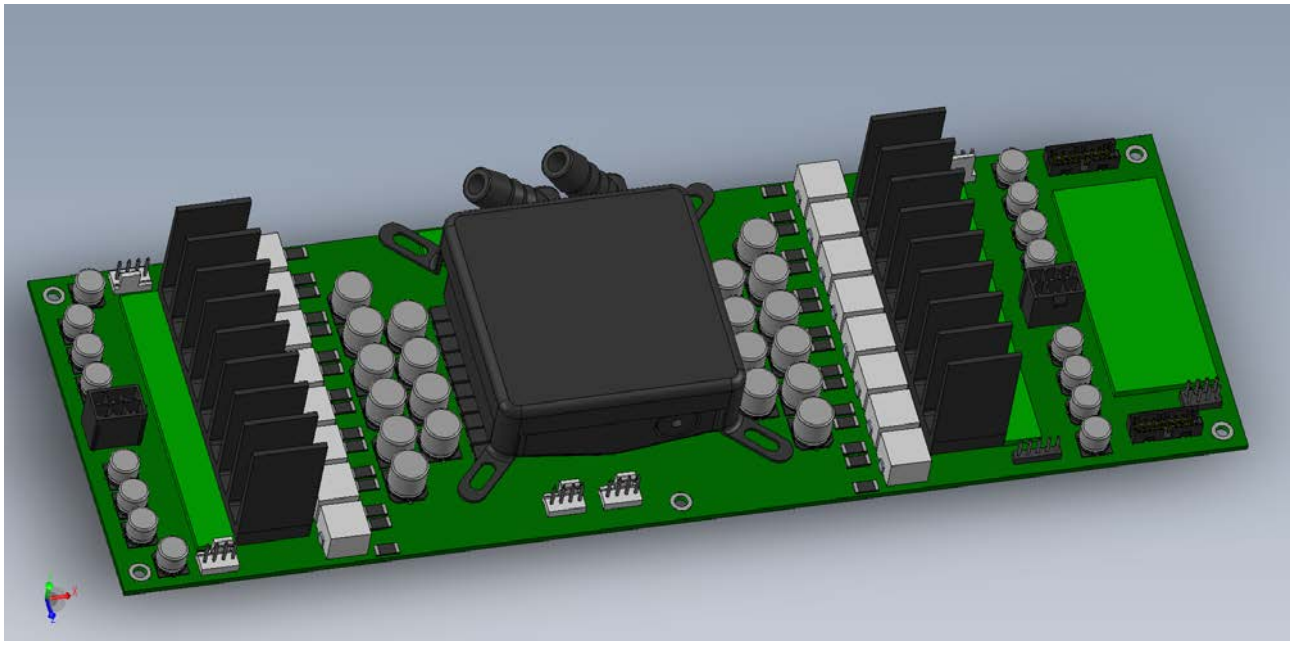


Figure 4.1: Basic HashFast GN module assembly

Tubing for connection to the separately mounted radiator unit is not shown, but the connections can be seen at the top of the head unit assembly.

When several modules are chained, only one module has a USB connection actually used, and that USB connection controls all modules in the chain. We denote this module as the “master” module. The “USB” connector on other modules is inactive.

Physical constraints and power distribution considerations tend to set limits on how many modules one might want to chain together. Architecturally, we could chain hundreds of modules together, but this would not be a very practical arrangement.

In the case of the initial Sierra rack mount product, three modules are chained together, as illustrated in Figure 4.2.

Some deployments may include an internal embedded CPU module to act as a host. Other deployments may use an external PC or rack mounted 0.5U or 1U server. In all cases the underlying interface to the internal module assembly is USB.

Third party vendors who realize their own design based on the HashFast ASIC may choose to use a similar interface, or may choose alternative means. Ultimately, the interface with the ASICs is serial in nature, but this fact will generally be internal and an interface presented to the user will operate at a higher level of abstraction, either USB or Ethernet with an embedded controller.

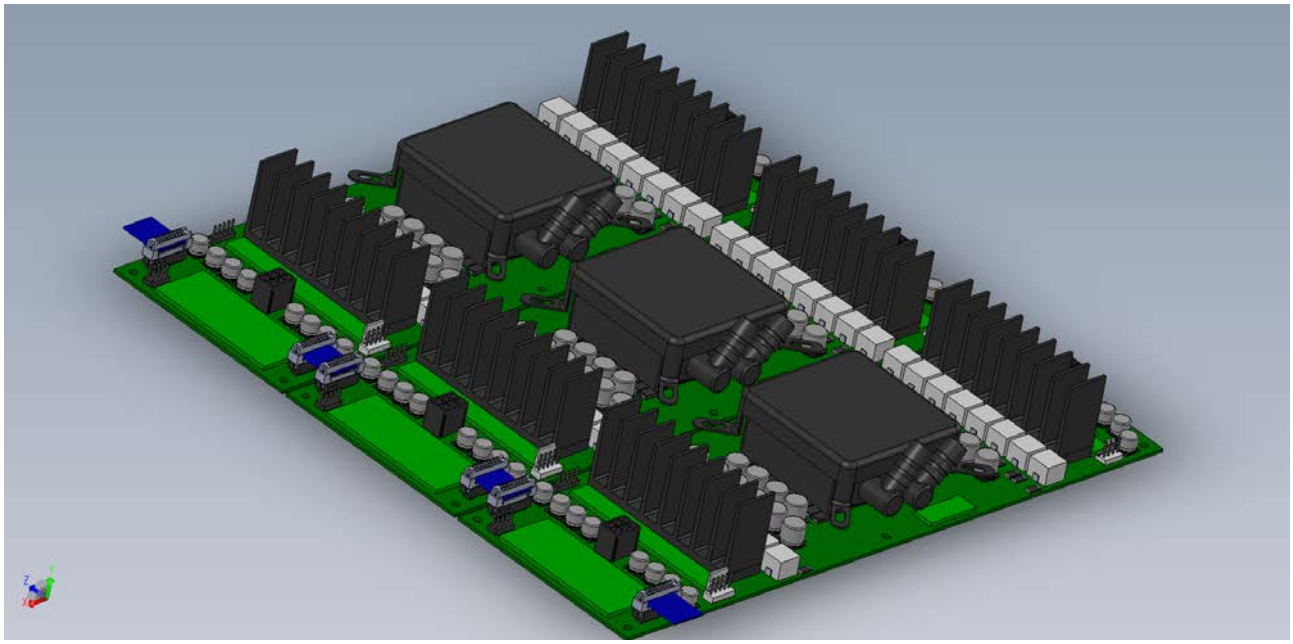


Figure 4.2: Three HashFast GN modules daisy chained.

APPENDICES

A Optimal Firmware/Hardware design notes

On Sep 15 2013, a posting entitled “Optimal Firmware/Hardware design for mining with cgminer” was made on the bitcointalk forum, here - https://bitcointalk.org/index.php?topic=294499.0 – with a couple of subsequent additions. This appendix contains a copy of version 0.3 of these guidelines, with notes to each bullet item as to how that aspect is handled by the HashFast GN ASIC and associated mining products.

With our thanks to “kano” for publishing this interesting and well considered list. Our notes below are in italicized text, the original guidelines are in smaller font.

- Firstly, have a firmware version string available. This clears up any issues about code handling changes to the firmware ... as long as you change the version string when the firmware changes. Even better to have something like the *** GetInfo that returns the version string and other relevant device configuration information, that cgminer would get when it initialises the device. If you are making multiple devices that cgminer has to treat differently, then the 'GetInfo' idea is a must.

  HashFast products include ASIC revision level, Module firmware level, Board revision level, serial number, and comprehensive system configuration information as a response to the initial OP_USB_INIT exchange used to initialize a device. See section 3.3.1 on page 42 for details

- Make sure it has a well defined iProduct and/or iManufacturer code so cgminer can tell straight away if it is a device of interest rather than not knowing until it has done cgminer code I/O to it. USB chips are used by many different devices, cgminer can simply ignore a device based on iProduct and/or iManufacturer - as it already does for some devices.

  HashFast is a member of the USB Interface Forum, and as such we have been issued with our own unique USB Vendor ID – 0x279c.

- Each device you produce should have a different iSerial - i.e. something that will identify each device differently when a person has more than one of them.

  Every HashFast product will have a unique iSerial number

- If you are considering improving the firmware in the future (who isn't?) - then being able to upload the firmware via USB is pretty much mandatory and thus making it possible for people who bought the devices early, to update their firmware and not be left behind due to being the early adopters.

  Firmware can be upgraded in HashFast products directly over the USB link. No physical interaction with the device is required to perform this update, it can be initiated remotely if required, and the HashFast supplied utility can update multiple devices at a time from a single invocation of the update utility.

- Most important for mining, they need to have an input and output queue i.e. be able to send more than one work item, and the device queues up those items, moving onto the next one each time it completes one. This removes all USB latency for any work other than when there is an LP The size of the input queue should allow it to run for at

least 100ms, preferably longer. So basically what the cgminer code would do: (as I already do in the *** **) is have 2 threads, one feeding work into the device at the rate required to keep the queue from getting empty, and another to get the output queued results.

Work input, and nonce+status output, are completely asynchronous and separate queues in HashFast products. Regardless of which protocol is in use, the host always knows how much work can be queued to the HashFast device, and this work is dispatched in real time to the target hash cores once it is received over the USB link. Each hash core has space for an active job and a pending job, so changeover from active to pending is seamless and there are no unused hash clock cycles.

When using the Global Work Queue protocol (see section 3.4 on page 50), a user may elect to perform some work derivation within the mining product, so the number of work items required may be cut dramatically. The amount of work derivation is host controllable, but typically a factor of 24 or 32 would be used, so a 1.2 TH/sec product would only require the same amount of work that a 50GH/sec product would require from the host. This is a trivial amount of work for the host to provide.

- With an input work queue, It is also ideal to be able to send multiple work items at once. If the device has multiple processors that each require their own work item, then when all processors are idle, or when an LP occurs, it is the priority of cgminer to get work to all processors as quickly as possible, thus being able to send at least as many work items as processors, in a single USB transfer, ensures minimum time wasted for all processors This of course also requires being able to send multiple work items to multiple processors in the same USB transfer. Also, of course, if the device is able to process many work items in a short time frame, then it is ideal to be able to send many work items per USB transfer.

  Because all hash cores in HashFast products are double buffered and contain a pending work item, it is only necessary for the host to keep the pending work slots full in order to ensure that all hash cores are busy 100% of the time. Each search core contains two interleaved double hash engines, so at 550 Mhz, it takes 3.9 seconds for a search core to process a full 2^{32} nonce range. This is the amount of time available to the host to fill the pending work slot.

  HashFast products do allow bulk transfer of work items, up to 8 items at a time, but between the double buffering available in each core, and the large reduction in work required from the host when work is being derived internal to the HashFast product, bulk transfers are hardly necessary.

- The process of sending and receiving work should include an identifier, so that the work replies only need to return that identifier, not the whole work item, to identify the work item the results are from e.g. maybe 4 characters a-zA-Z0-9 so >14million before it recycles - bigger if your device hashes at more than 10TH/s - at 10TH/s 4 characters = ~105 minutes of work

  HashFast products identify work with a unique sequence number, and that sequence number is returned to the host along with any nonces that are found. The rolling head and tail sequence numbers allow the host to determine when work has been completed, how much work is in flight, and how much additional work can be queued to the device.

- Results should be put in the output queue when they are found, not waiting for the nonce range to complete. Thus, you also need to send a 'finished' work item result since ~1/3 of work has no nonces. However, in the further future when a nonce range takes less than 1ms, it won't really be very relevant if the results aren't queued in the output queue until the nonce range is completed.

  In HashFast products, nonces are returned to the host immediately after they are found, there is no delay and no waiting for the associated job to complete – the sequence number of the job is returned along with the nonce to allow the host to identify the associated work while it is still in progress.

  HashFast products do not send a "work finished"notification to the host. Since cores are double buffered, there is no particular urgency for the host to replace a work item, hash cores are never idle when managed properly in this manner. In the case of the UMS protocol, the sequence number and core status bitmap allow the host to determine when a job has completed and to replace it for a particular hash core. For the GWQ protocol, the host has no knowledge of individual core activity, it only has to feed work into a Global Work Queue in accordance with a simple calculation of how much work is needed at any particular time, and to free work items by inference in accordance with the device tail sequence pointer.

  There is an inherent robustness in this mechanism. Even though USB communication is reliable, if for some reason a periodic status message went missing, the problem would be naturally recovered as the following status message(s) came back – the tail pointer would catch up. If a work message went missing, again the tail pointer would progress past the missing work and the host would recover.

  Finally, there is no motivation to shortening the nonce range in order to drop work time down to fractions of a second, because HashFast products allow work to be aborted, and in particular with the GWQ protocol, the OP_WORK_RESTART mechanism provides a built in command to seamlessly abort all work in progress and replace it with new work based on a new coin base, as often as required and with negligible hashing downtime

- The device needs 3 types of reset:
  1) to reset the work on an LP: clear the input queue, abort current work and accept new work to start on - preferably all in one command
  2) to stop work due to e.g. overheating: clear the input queue and stop work
  This of course also needs reliable temperature sensors
  3) to clear the input queue, queue new work, but finish current work - like the LP but where the current work is allowed to complete ... e.g. a difficulty change being received by cgminer, providing new work at the new difficulty

  (1) We expect the GWQ protocol to be generally used for HashFast products, as such the OP_WORK_RESTART mechanism described in section 3.4.2 on page 53 will be used to switch in fresh work seamlessly as Stratum servers deman work restarts. If for some reason the UMS protocol is being used, work can be aborted on a per core (and thereby a per-sequence number) basis by the host using the OP_ABORT function.

  (2) When using the GWQ protocol, HashFast products perform thermal management internally by load shedding and restoration. If a host attempts to over-clock the ASICs excessively, cores

will be run down internally to maintain adequate thermal margin. The host will be advised about the extent of this as part of the regular status reporting. When using the UMS protocol, the host may perform thermal management by aborting work on cores to shed load.

Note that HashFast products have a thermal overload limiter, which will result in a complete system shutdown, but this operates at an emergency temperature level, whereas load shedding to perform normal thermal management operates below this level.

(3) Since the "search difficulty" is individually specified for each OP_HASH work item queued to HashFast devices, and individual cores can be working at different levels of difficulty from each other, it is never necessary to perform any sort of work run down associated with a server difficulty change – just apply the new difficulty to all new work items.

- The combination of the above 2 - nonce answers immediately and being able to abort work - resolves any issues with short LP times e.g. P2pool
- Regarding overheating ... the hardware should only shutdown mining at some critical self destruct level - like GPUs do. cgminer should control this below that critical level at the user's chosen temperature constraints. Again (as above) this of course also needs reliable temperature sensors - and the hardware developer to provide details of how to interpret the results of the sensors

The thermal overload mechanism in HashFast products, as described in section 2.4.2.6, is just that – an emergency shutdown mechanism to avoid equipment damage. Below this point, mining software can perform thermal control, and there are ample temperature sensors provided; however, for the GWQ protocol, the HashFast device itself will perform thermal control internally between limits that may be set by the user.

HashFast products contain both on-die temperature sensors and board level temperature sensors, as described in this user guide.

- It's optimal to have asynchronous processing - send something to be done but not expect the reply to be the next thing returned. This can easily be achieved with two things: 1) each command that is expecting a reply should send an identifier with it and 2) the replies would be placed in the output queue with the identifier that was sent. Thus only one thread is dealing with reading replies, and waiting for status replies doesn't get in the way of getting results

Except for the one-time power-up device initialization, where there is only one request-reply pair, all communication in each direction is asynchronous and received traffic is processed on its own merit. Where an identifier is required to match received information with something that was previously sent (such as relating a nonce to a previously sent work item), sequence numbers serve as unique identifiers to allow received traffic to be related to the corresponding sent traffic.

In some cases a host may wish to send something to all ASICs, such as a global abort, and know when it has completed. Broadcast addressing capabilities allow this to be done, and moreover the host will receive back the frame it sent, in some cases possibly modified, but again the content of the receive frame can be related to what was originally sent by lieu of fields contained within the frame itself.

- One thing I have thought might be good, but I'm not sure of the MCU implementation issues, would be to have a USB chip with at least 2 pairs of end points, the 2nd one being purely for results, so the results thread simply waits on replies rather than polling for them

  The USB controller used in HashFast products allows up to 16 endpoints to be defined. CGMiner now uses asynchronous USB communications, and there does not seem to be any great benefit from using multiple endpoints. Our current lab models use a single endpoint pair for writing and reading. However, the USB controller can do ping-pong buffering when separate endpoint pairs are used for each communication direction, and we will evaluate such a configuration to see what benefits, if any, this yields.

  Again, since the GWQ protocol can do work derivation/multiplication within the hardware, and since the product can operate at any practical level of difficulty (thereby keeping nonce traffic down to reasonable limits), USB communication demands with HashFast products are actually quite modest.

- Related to this, of course you should be able to send status requests while it is mining, but that really only becomes an issue when you don't have a queue, so it shouldn't be relevant (i.e. it should always be implements to allow sending status requests while mining)

  Status information is regularly sent to the host as part of the HashFast protocols. The frequency of this, and whether or not it occurs as an event driven mechanism, is completely configurable.

- As the devices get faster, also allow difficulty to be defined in the work sent, so it will only return nonces at the requested difficulty. HOWEVER, you'd want to be sure there's no loss of normal or error information - e.g. if a nonce is passed to the MCU difficulty checker, and there is a problem with it, it shouldn't hide that fact, but rather report that fact to the miner - a hardware error Of course related to what I mentioned before, you need to send information saying the nonce ranges are complete so that information isn't lost about how much work was done. Maybe as devices get faster (in the future) this complete message could be one reply with a list of work items that have completed

  Every hash job submitted to a HashFast product can have difficulty set to anywhere from Diff 1 up to ridiculously high levels – see section 2.4.2.1 on page 25 for the internal limits. All nonces found are passed to the host, so if there is a HW error (such as might occur with excessive over-clocking), the host can log it.

  The UMS protocol's OP_STATUS message is in fact a single reply (per die) with information about all work status of all cores; this along with the sequence number for which the core status is valid. This allows the host to determine when jobs have completed, across all cores.

  However, since we recommend using the higher level GWQ protocol, the host need not be aware of individual core activity (or even of how many cores are out there). Determination of when jobs have completed is made by monitoring the tail sequence pointer, which is analogous to a rolling sequence identifyingbatches of jobs that have completed.

  The GWQ protocol is inherently scale-able, and the mechanisms it uses work for the largest devices we are assembling today or expect to have for the foreseeable future.

B CRC-8 reference, and sample packet functions

We provide here an example of the trivial code required to perform CRC-8 generation and checking, along with sample C code associated with generating, sending and receiving transactions. For bulk transmission of multiple work items, a user is more likely to construct the bulk data locally in a buffer, and call the USB write interface function directly (not shown).

```
////////////////////////////////////////////////////////////////////////////
// Support for the CRC-8 used in the header
////////////////////////////////////////////////////////////////////////////
#define GP8 0x107  /* x^8 + x^2 + x + 1 */
#define DI8 0x07

static unsigned char crc8_table[256];  /* CRC-8 table */
static int made_crc8_table=0;

static void hf_init_crc8()
  {
  int i, j;
  unsigned char crc;

  if (!made_crc8_table) {
    for (i=0; i<256; i++) {
      crc = i;
      for (j=0; j<8; j++)
        crc = (crc << 1) ^ ((crc & 0x80) ? DI8 : 0);
      crc8_table[i] = crc & 0xFF;
      }
    made_crc8_table=1;
    }
  }

static unsigned char hf_crc8(unsigned char *h)
  {
  int i;
  unsigned char crc;

  if (!made_crc8_table)
    hf_init_crc8();
  h++;            // Preamble not included
  for (i=1, crc=0xff; i < 7; i++)
    crc = crc8_table[crc ^ *h++];
  return(crc);
  }

static int hashfast_get_header(struct cgpu_info *, struct hf_header *, uint8_t *);
static int hashfast_get_data(struct cgpu_info *, uint8_t *, int);
static int hashfast_get_packet(struct cgpu_info *, struct hf_header *);
static int hashfast_send_frame(struct cgpu_info *, uint8_t, uint8_t, uint8_t, uint16_t, uint8_t *, int);

////////////////////////////////////////////////////////////////////////////
// Read an 8 byte header, or timeout
//
// Strictly speaking this should have a start-of-frame search mode when we are out of
// frame sync, i.e. look for a candidate preamble, 1 byte at a time. We get away
// without it because protocol traffic is infrequent/sporadic, so most of the
// time the line is idle.
//
////////////////////////////////////////////////////////////////////////////
static int hashfast_get_header(struct cgpu_info *hashfast, struct hf_header *h, uint8_t *computed_crc)
  {
  int read_amount = sizeof(*h);
  ssize_t ret = 0, offset = 0;
  int retries = 0;
  int amount;

  do {
    amount = 0;
    ret = usb_read_timeout(hashfast, (char *)h, read_amount, &amount, 1, 0);
    if (ret < 0 || amount == 0)
      cgsleep_ms(20);
    } while (amount == 0 && ++retries < 100);
  if (amount == 0)
    return(0);                      // Timeout
  offset = (ssize_t) amount;
  if (amount != sizeof(*h))
    {
    applog(LOG_WARNING, "HF%d: get_header: Strange amount returned %d", hashfast->device_id, amount);
```

```
    return(offset);
    }
  *computed_crc = hf_crc8((uint8_t *)h);
  return(offset);
  }

////////////////////////////////////////////////////////////////////////////
// Get the body section of a "long" packet. Only called after we have already
// validated a header.
////////////////////////////////////////////////////////////////////////////

static int hashfast_get_data(struct cgpu_info *hashfast, uint8_t *buf, int len4)
  {
  int read_amount = len4 * 4;
  ssize_t ret = 0, offset = 0;
  int amount;
  int retries;
  int timeout;
  uint8_t *p;

  memset(buf, 0, read_amount);
  retries = 0;

  do {
    amount = 0;
    ret = usb_read_once_timeout(hashfast, buf, read_amount, &amount, 1, 0);
    if (ret < 0 || amount == 0)
      cgsleep_ms(20);
    } while (amount == 0 && ++retries < 100);
  if (amount == 0)
    return(0);                // Timeout
  offset = (ssize_t) amount;
  if (amount != read_amount)
    {
    applog(LOG_WARNING, "HF%d: get_data: Strange amount returned %d vs. expected %d", hashfast->device_id, amount, read_amount);
    return(offset);
    }
  return(len4);
  }

////////////////////////////////////////////////////////////////////////////
// Get an entire packet, header and optional data
////////////////////////////////////////////////////////////////////////////
static int hashfast_get_packet(struct cgpu_info *hashfast, struct hf_header *h)
  {
  uint8_t hcrc;
  int ret;

  if ((ret = hashfast_get_header(hashfast, h, &hcrc)) <= 0)
    return(ret);
  else if ((h->preamble != HF_PREAMBLE) || (h->crc8 != hcrc))
    {
    applog(LOG_NOTICE, "HF%d: bad header: B: ret %d, hcrc 0x%02x, reply:", hashfast->device_id, ret, hcrc);
    hexdump((uint8_t *)h, sizeof(*h));
    return(-1);
    }
  else
    {
    applog(LOG_DEBUG, "HF%d: got header", hashfast->device_id);
    hexdump((uint8_t *)h, sizeof(*h));
    }
  if (h->data_length > 0)
    {
    if (hashfast_get_data(hashfast, (uint8_t *)(h+1), h->data_length) != h->data_length)
      {
      applog(LOG_ERR, "HF%d: bad data section", hashfast->device_id);
      return(-1);
      }
    else
      {
      applog(LOG_DEBUG, "HF%d: got data (%d bytes):", hashfast->device_id, h->data_length*4);
      hexdump((uint8_t *)(h+1), h->data_length*4);
      }
    }

  return(ret + h->data_length*4);
  }

////////////////////////////////////////////////////////////////////////////
// Send an arbitrary frame, consisting of an 8 byte header and an optional packet body.
////////////////////////////////////////////////////////////////////////////
static int hashfast_send_frame(struct cgpu_info *hashfast, uint8_t opcode, uint8_t chip, uint8_t core, uint16_t hdata, uint8_t *data, int len)
  {
  uint8_t packet[256];
  struct hf_header *p = (struct hf_header *) packet;
  int tx_length;
  int ret;
```

```
  int id = hashfast->device_id;
  int amount;
  int retries;

  p->preamble = HF_PREAMBLE;
  p->operation_code = opcode;
  p->chip_address = chip;
  p->core_address = core;
  p->hdata = hdata;
  p->data_length = len / 4;
  p->crc8 = hf_crc8((uint8_t *)p);

  if (len)
    memcpy(&packet[sizeof(struct hf_header)], data, len);
  tx_length = sizeof(struct hf_header) + len;

  retries = 0;
  do
    {
    amount = 0;
    ret = usb_write_timeout(hashfast, (char *)packet, tx_length, &amount, 200, 0);
    } while ((amount != tx_length) && ++retries < 5);
  if (amount != tx_length)
    {
    applog(LOG_ERR, "HF%d: send_frame: USB Send error, ret %d amount %d vs. tx_length %d retry %d", id, ret, amount, tx_length, retries);
    return 1;
    }

  return 0;
  }

////////////////////////////////////////////////////////////////////////////////
// Send an already constructed header
////////////////////////////////////////////////////////////////////////////////
static int hashfast_send_header(struct cgpu_info *hashfast, struct hf_header *h)
  {
  int retries = 0;
  int amount;
  int ret;

  do
    {
    amount = 0;
    ret = usb_write_timeout(hashfast, (char *)h, sizeof(*h), &amount, 200, 0);
    } while ((amount != sizeof(*h)) && ++retries < 5);
  if (amount != sizeof(*h))
    {
    applog(LOG_ERR, "HF%d: send_header: USB Send error, ret %d amount %d vs. length %d retry %d", hashfast->device_id, ret, amount,
sizeof(*h), retries);
    return 1;
    }

  return 0;
  }
```

C HashFast provided header file – hf_protocol.h

The following header file includes useful structure definitions, and may be included directly by device driver developers. This file is available in electronic form from HashFast and is licensed under GPLv3:

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

```
//
// Copyright 2013 HashFast LLC
//
// This program is free software; you can redistribute it and/or modify it
// under the terms of the GNU General Public License as published by the Free
// Software Foundation; either version 3 of the License, or (at your option)
// any later version. See COPYING for more details.
//
// Useful data structures and values for interfacing with HashFast products
//


#ifndef _HF_PROTOCOL_H_
#define _HF_PROTOCOL_H_

#if __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
  #error "This header uses bit fields and has byte ordering assumptions suitable only for Little Endian platform
s"
#endif

#define HF_PREAMBLE             (uint8_t) 0xaa
#define HF_BROADCAST_ADDRESS    (uint8_t) 0xff
#define HF_GWQ_ADDRESS          (uint8_t) 254

// Serial protocol operation codes (Second header byte)
#define OP_ROOT         1
#define OP_RESET        2
#define OP_PLL_CONFIG   3
#define OP_ADDRESS      4
#define OP_READDRESS    5
#define OP_HIGHEST      6
#define OP_BAUD         7
#define OP_UNROOT       8

#define OP_HASH         9
#define OP_NONCE        10
#define OP_ABORT        11
#define OP_STATUS       12
#define OP_GPIO         13
#define OP_CONFIG       14
#define OP_STATISTICS   15
#define OP_GROUP        16
#define OP_CLOCKGATE    17

// Generic header
struct hf_header {
        uint8_t  preamble;                      // Always 0xaa
        uint8_t  operation_code;
        uint8_t  chip_address;
        uint8_t  core_address;
        uint16_t hdata;                         // Header specific data
        uint8_t  data_length;                   // .. of data frame to follow, in 4 byte blocks, 0=no data
        uint8_t  crc8;                          // Computed across bytes 1-6 inclusive
```

```
        } __attribute__((packed,aligned(4)));   // 8 bytes total

// Header specific to OP_PLL_CONFIG
struct hf_pll_config {
        uint8_t  preamble;
        uint8_t  operation_code;
        uint8_t  chip_address;

        uint8_t  pll_divr:6;
        uint8_t  pll_bypass:1;
        uint8_t  pll_reset:1;

        uint16_t pll_divf:8;
        uint16_t spare1:1;                      // Must always be 0
        uint16_t pll_divq:3;
        uint16_t pll_range:3;
        uint16_t pll_fse:1;                     // Must always be 1

        uint8_t  data_length;                   // Always 0
        uint8_t  crc8;                          // Computed across bytes 1-6 inclusive
        } __attribute__((packed,aligned(4)));   // 8 bytes total

// OP_HASH serial data
struct hf_hash_serial {
        uint8_t  midstate[32];                  // Computed from first half of block header
        uint8_t  merkle_residual[4];            // From block header
        uint32_t timestamp;                     // From block header
        uint32_t bits;                          // Actual difficulty target for block header
        uint32_t starting_nonce;                // Usually set to 0
        uint32_t nonce_loops;                   // How many nonces to search, or 0 for 2^32
        uint16_t ntime_loops:12;                // How many times to roll timestamp, or 0
        uint16_t spare1:4;
        uint8_t  search_difficulty;             // Search difficulty to use, # of '0' digits required
        uint8_t  option;
        uint32_t group:8;
        uint32_t spare3:24;
        uint32_t crc32;                         // Computed across all preceding data fields
        } __attribute__((packed,aligned(4)));   // 64 bytes total, including CRC

// OP_HASH usb data - squashed so header+data = 64 bytes
struct hf_hash_usb {
        uint8_t  midstate[32];                  // Computed from first half of block header
        uint8_t  merkle_residual[4];            // From block header
        uint32_t timestamp;                     // From block header
        uint32_t bits;                          // Actual difficulty target for block header
        uint32_t starting_nonce;                // Usually set to 0
        uint32_t nonce_loops;                   // How many nonces to search, or 0 for 2^32
        uint16_t ntime_loops:12;                // How many times to roll timestamp, or 0
        uint16_t spare1:4;
        uint8_t  search_difficulty;             // Search difficulty to use, # of '0' digits required
        uint8_t  group;                         // Non-zero for valid group
        } __attribute__((packed,aligned(4)));   //
```

```
// OP_NONCE data
struct hf_candidate_nonce {
        uint32_t nonce;                         // Candidate nonce
        uint16_t sequence;                      // Sequence number from corresponding OP_HASH
        uint16_t ntime:12;                      // ntime offset, if ntime roll occurred
        uint16_t search:1;                      // Search forward next 128 nonces to find solution
        uint16_t spare:3;
        } __attribute__((packed,aligned(4)));

// OP_CONFIG data
struct hf_config_data {
        uint16_t status_period:11;                  // Periodic status time, msec
        uint16_t enable_periodic_status:1;          // Send periodic status
        uint16_t send_status_on_core_idle:1;        // Schedule status whenever core goes idle
        uint16_t send_status_on_pending_empty:1;    // Schedule status whenever core pending goes idle
        uint16_t pwm_active_level:1;                // Active level of PWM outputs, if used
        uint16_t forward_all_privileged_packets:1;  // Forward priv pkts -- diagnostic
        uint8_t  status_batch_delay;                // Batching delay, time to wait before sending status
        uint8_t  watchdog:7;                        // Watchdog timeout, seconds
        uint8_t  disable_sensors:1;                 // Diagnostic

        uint8_t  rx_header_timeout:7;               // Header timeout in char times
        uint8_t  rx_ignore_header_crc:1;            // Ignore rx header crc's (diagnostic)
        uint8_t  rx_data_timeout:7;                 // Data timeout in char times / 16
        uint8_t  rx_ignore_data_crc:1;              // Ignore rx data crc's (diagnostic)
        uint8_t  stats_interval:7;                  // Minimum interval to report statistics (seconds)
        uint8_t  stat_diagnostic:1;                 // Never set this
        uint8_t  measure_interval;                  // Die temperature measurement interval (msec)

        uint32_t one_usec:12;                       // How many LF clocks per usec.
        uint32_t max_nonces_per_frame:4;            // Maximum # of nonces to combine in a single frame
        uint32_t voltage_sample_points:8;           // Bit mask for sample points (up to 5 bits set)
        uint32_t pwm_phases:2;                      // phases - 1
        uint32_t trim:4;                            // Trim value for temperature measurements
        uint32_t clock_diagnostic:1;                // Never set this
        uint32_t forward_all_packets:1;             // Forward everything - diagnostic.

        uint16_t pwm_period;                        // Period of PWM outputs, in reference clock cycles
        uint16_t pwm_pulse_period;                  // Initial count, phase 0
        } __attribute__((packed,aligned(4)));

// OP_GROUP data
struct hf_group_data {
        uint16_t nonce_msoffset;                    // This value << 16 added to starting nonce
        uint16_t ntime_offset:12;                   // This value added to timestamp
        uint16_t spare:4;
        } __attribute__((packed,aligned(4)));

// Structure of the monitor fields for G-1, returned in OP_STATUS, core bitmap follows this
struct hf_g1_monitor {
        uint16_t die_temperature:12;                // Die temperature ADC count
        uint16_t spare:4;                           // Spare
        uint8_t  core_voltage[6];                   // Core voltage
```

```
                                                // [0] = main sensor
                                                // [1]-[5] = other positions
        } __attribute__((packed,aligned(4)));

// Conversions for the ADC readings from GN on-chip sensors in the above structure
#define GN_CORE_VOLTAGE(a)              ((float)(a)/256*1.2)
#define GN_DIE_TEMPERATURE(a)           ((((float)(a)*240)/4096.0)-61.5)

// What comes back in the body of an OP_STATISTICS frame (On die statistics)
struct hf_statistics {
        uint8_t rx_header_crc;                  // Header CRC error's
        uint8_t rx_body_crc;                    // Data CRC error's
        uint8_t rx_header_timeouts;             // Header timeouts
        uint8_t rx_body_timeouts;               // Data timeouts
        uint8_t core_nonce_fifo_full;           // Core nonce Q overrun events
        uint8_t array_nonce_fifo_full;          // System nonce Q overrun events
        uint8_t stats_overrun;                  // Overrun in statistics reporting
        uint8_t spare;
        } __attribute__((packed,aligned(4)));


// The sequence distance between a sent and received sequence number.
#define SEQUENCE_DISTANCE(tx,rx)        ((tx)>=(rx)?((tx)-(rx)):(info->num_sequence+(tx)-(rx)))

////////////////////////////////////////////////////////////////////////////
// USB protocol data structures
////////////////////////////////////////////////////////////////////////////

// Convenience header specific to OP_USB_INIT
struct hf_usb_init_header {
        uint8_t  preamble;                      // Always 0xaa
        uint8_t  operation_code;
        uint8_t  spare1;

        uint8_t  protocol:3;                    // Which protocol to use
        uint8_t  user_configuration:1;          // Use the following configuration data
        uint8_t  spare2:4;

        uint16_t hash_clock:12;                 // Requested hash clock frequency
        uint16_t pll_bypass:1;                  // Force PLL bypass, hash clock = ref clock
        uint16_t no_asic_initialization:1;      // Do not perform automatic ASIC initialization
        uint16_t do_atspeed_core_tests:1;       // Do core tests at speed, return second bitmap
        uint16_t leave_powered_down:1;          // Init USB only, leave device powered down

        uint8_t  data_length;                   // .. of data frame to follow, in 4 byte blocks
        uint8_t  crc8;                          // Computed across bytes 1-6 inclusive
        } __attribute__((packed,aligned(4)));   // 8 bytes total

// Values the protocol field in the above structure may take
#define PROTOCOL_USB_MAPPED_SERIAL      0
#define PROTOCOL_GLOBAL_WORK_QUEUE      1

// Options (only if present) that may be appended to the above header
```

```
// Each option involving a numerical value will only be in effect if the value is non-zero
// This allows the user to select only those options desired for modification. Do not
// use this facility unless you are an expert - loading inconsistent settings will not work.
struct hf_usb_init_options {
        uint16_t group_ntime_roll;                  // Total ntime roll amount per group
        uint16_t core_ntime_roll;                   // Total core ntime roll amount
        uint8_t  low_operating_temp_limit;          // Lowest normal operating limit
        uint8_t  high_operating_temp_limit;         // Highest normal operating limit
        uint16_t spare;
        } __attribute__((packed,aligned(4)));

// Base item returned from device for OP_USB_INIT
struct hf_usb_init_base {
        uint16_t firmware_rev;                      // Firmware revision #
        uint16_t hardware_rev;                      // Hardware revision #
        uint32_t serial_number;                     // Board serial number
        uint8_t  operation_status;                  // Reply status for OP_USB_INIT (0 = success)
        uint8_t  extra_status_1;                    // Extra reply status information, code specific
        uint16_t sequence_modulus;                  // Sequence numbers are to be modulo this
        uint16_t hash_clockrate;                    // Actual hash clock rate used (nearest Mhz)
        uint16_t inflight_target;                   // Target inflight amount for GWQ protocol
        } __attribute__((packed,aligned(4)));

// The above base item (16 bytes) is followed by the struct hf_config_data (16 bytes) actually
// used internally (so users may modify non-critical fields by doing subsequent
// OP_CONFIG operations). This is followed by a device specific "core good" bitmap (unless the
// user disabled initialization), and optionally by an at-speed "core good" bitmap.


// Information in an OP_DIE_STATUS frame. This is for one die - there are four per ASIC.
// Board level phase current and voltage sensors are likely to disappear in later production models.
struct hf_g1_die_data {
        struct hf_g1_monitor die;                   // Die sensors - 8 bytes
        uint16_t phase_currents[4];                 // Phase currents (0 if unavailable)
        uint16_t voltage;                           // Voltage at device boundary (0 if unavailable)
        uint16_t temperature;                       // Regulator temp sensor
        uint16_t tacho;                             // See documentation
        uint16_t spare;
        } __attribute__((packed,aligned(4)));       // 24 bytes total

// Conversions for the board/module level sensors
#define M_VOLTAGE(a)                    ((float)(a)*19.0734e-6)
#define M_PHASE_CURRENT(a)              ((float)(a)*0.794728597e-3)

// Information for an OP_GWQ_STATUS frame
// If sequence_head == sequence_tail, then there is no active work and sequence_head is invalid
struct hf_gwq_data {
        uint64_t hash_count;                        // Add this to host's cumulative hash count
        uint16_t sequence_head;                     // The latest, internal, active sequence #
        uint16_t sequence_tail;                     // The latest, internal, inactive sequence #
        uint16_t shed_count;                        // # of cores have been shedded for thermal control
        uint16_t spare;
        } __attribute__((packed,aligned(4)));
```

```
// Information for an OP_USB_STATS1 frame - Communication statistics
struct hf_usb_stats1 {
        // USB incoming
        uint16_t usb_rx_preambles;
        uint16_t usb_rx_receive_byte_errors;
        uint16_t usb_rx_bad_hcrc;

        // USB outgoing
        uint16_t usb_tx_attempts;
        uint16_t usb_tx_packets;
        uint16_t usb_tx_timeouts;
        uint16_t usb_tx_incompletes;
        uint16_t usb_tx_endpointstalled;
        uint16_t usb_tx_disconnected;
        uint16_t usb_tx_suspended;

        // Internal UART transmit
        uint16_t uart_tx_queue_dma;
        uint16_t uart_tx_interrupts;

        // Internal UART receive
        uint16_t uart_rx_preamble_ints;
        uint16_t uart_rx_missed_preamble_ints;
        uint16_t uart_rx_header_done;
        uint16_t uart_rx_data_done;
        uint16_t uart_rx_bad_hcrc;
        uint16_t uart_rx_bad_dma;
        uint16_t uart_rx_short_dma;
        uint16_t uart_rx_buffers_full;

        uint8_t  max_tx_buffers;                    // Maximum # of send buffers ever used
        uint8_t  max_rx_buffers;                    // Maximum # of receive buffers ever used
        } __attribute__((packed,aligned(4)));

// Values info->device_type can take
#define HFD_G1                          1       // HashFast G-1 GN ASIC
#define HFD_VC709                     128
#define HFD_ExpressAGX                129

// USB interface specific operation codes
#define OP_USB_INIT                   128       // Initialize USB interface details
#define OP_GET_TRACE                  129       // Send back the trace buffer if present
#define OP_LOOPBACK_USB               130
#define OP_LOOPBACK_UART              131
#define OP_DFU                        132       // Jump into the boot loader
#define OP_USB_SHUTDOWN               133       // Initialize USB interface details
#define OP_DIE_STATUS                 134       // Die status. There are 4 die per ASIC
#define OP_GWQ_STATUS                 135       // Global Work Queue protocol status
#define OP_WORK_RESTART               136       // Stratum work restart regime
#define OP_USB_STATS1                 137
#define OP_USB_GWQSTATS               138
```

```
#define OP_USB_DEBUG                    255

// HashFast vendor and product ID's
#define HF_USB_VENDOR_ID                0x297c
#define HF_USB_PRODUCT_ID_G1            0x0001

//
// Fault codes that can be returned in struct hf_usb_init_base.operation_status
//
#define E_RESET_TIMEOUT                 1
#define E_ADDRESS_TIMEOUT               2
#define E_CLOCKGATE_TIMEOUT             3
#define E_CONFIG_TIMEOUT                4

#define U32SIZE(x)                      (sizeof(x)/sizeof(uint32_t))


#endif
```