Iridium Packet Type ‘0’

Used for Spray2 SBD and BGC Rudics Jul2021

Type ‘0’ (zero) replaces type ‘X’ for a message block identifier, allowing more header bytes for each sensor packet. Details can be found at

G:\Shared drives\Spray2\docs\software\serverCode\MessageBlockFormat.gdoc

Link

Some of that info will be repeated here.

Sensor Header

Each sensor packet now has a 6-byte header, instead of the old 3-bytes. The header of each sensor data packet looks like:

<sensorID>	**(**one byte)
<Ver> <nDatSize>	(2 bytes: MSN=Ver, 3 lower nibbles=nDatSize)
<packType><data_ID>	(new byte: one nibble each)
<proType><proDir>	(new byte: one nibble each)
<pSame><subBlk>	(new byte: one nibble each)

<sensorID> = [1 byte] =ID to identify the type of sensor family.
<ver><nDatSize> = [2 bytes]
<Ver>= MSNibble, allows for future mods to the data format.
<nDatSize> = 3 nibbles = number of bytes in the packet. This includes all the bytes in the packet, from <sensorID> to ‘;’.
<packType> =[byte #4, bits 4:7==MSN] = data-packing algorithm identifier.
<data_ID> =[byte #4, bits 0:3==LSN] = ID for each sensor’s data product/type.
Specific to profile series:
<proType> =[byte #5, bits 4:7==MSN] = flags for how the profile is averaged.
<proDir> =[byte #5, bits 0:3==LSN] = flags for how the profile is acquired.
<reserved> =[byte #6, bits 4:7==MSN]: Presently not used.
<subSeg> =[byte #6, bits 0:3==LSN] =profile sub-segment ID (used by Argo floats).
Non-profiling packets can use these last two bytes for other info.
Specific to profile series, Added 11Oct21.
These extra profile-header bytes are **not** present for non-profiling packets.
<kPro> =[byte #7, 8] = index into the ideal full profile for **<subSeg>**
<numPro> =[byte #9,10] = total number of scans in the profile.
More explanation is provided below.

The above sensor header is then followed by the data:
<sensorData> = (<nDatSize> - 7) bytes of data, packed via **<packType>** algorithm.
‘;’ = delimiter at the end of each sensor packet.

Definition of Sensor Header Parameters

<sensor_ID>

Identifies a certain sensor, which is assumed to have a family of data products. For example, the GPS family includes “products” that are really the divePhase of the GPS fix. Sensors include the GPS, CTD, and each of the BGC sensors. See PktDef.h for definitions.

<Ver>

Allows future modifications to the sensor packet to be identified by the version.

<packType>

This is a detailed description of the <packType> nibble as applied to the BGC SOLO. The first 2 bits define **p1Type** and the 2nd 2 bits define **p2Type**. Both have value ranges of 0..3.

p1Type = 0 = no data compression, integer values.
p2Type+1 = (1..4) = number of bytes per datum.
p1Type = 1 = first-diff algorithm
p2Type = 0 == original SOLO algorithm.
p2Type = 1 == BGC (possibility of badBits in each sub-packet).
p2Type = 2 == BGC + auto-offset+gain for each sub-packet.
p2Type = 3 not used.
p1Type = 2 = 2nd-diff algorithm
p2Type = 0 == original SOLO algorithm.
p2Type = 1 == BGC (possibility of badBits in each sub-packet).
p2Type = 2 == BGC + auto-offset+gain for each sub-packet.
p2Type = 3 not used.
p1Type = 3 = other algorithms
p2Type = 0 not used
p2Type = 1 == packed as 4-byte float values.
p2Type = 2 not used.
p2Type = 3 not used.

In c code, the above straight-byte pack values can be defined as

```
#define pack_1byte 0x00 // pack_none + ( 0 * 4 ) =pack none, 1bytes ea
#define pack_2byte 0x04 // pack_none + ( 1 * 4 ) =pack none, 2bytes ea
#define pack_3byte 0x08 // pack_none + ( 2 * 4 ) =pack none, 3bytes ea
#define pack_4byte 0x0C // pack_none + ( 3 * 4 ) =pack none, 4bytes ea
#define pack_float 0x07 // pack_other==3 + 1*4 = 4-byte floats
```

<data_ID>

Identifies the data product/sensor variable in this data packet. For instance, the BGC ECO-Triplet has 3 channels. The <data_ID> identifies which channel is in this packet.

<proType> (presently only 2 bits are being used)

0x00 = data averaged in depth (normal for SOLO ascent/descent series).
0x01 = data averaged in time (normal for Spray, averaged every dt interval).
0x02 = raw data (no averaging: samples are separated in time, dt, typ. near-surface).
0x03 = data measured while in drift (discrete mode, proDir does not apply).

<proDir> (presently only 2 bits are being used)

0x00 = measured on ascent, continuous-mode. (normal for SOLO)
0x01 = measured on descent, continuous-mode.
0x02 = measured on ascent, discrete-mode.
0x03 = measured on descent, discrete-mode.

<reserved>

This nibble is used to further specify a certain sensor's data-set. For instance, as of Oct21 it is being used to identify the in-air DO time-series: DO in-air has <proType>=2, <proDir>=2 and <reserved> = 1.

<subSeg>

When a sensor's profile needs to be broken into multiple segments across Iridium blocks, <subSeg> is used to identify this particular profile segment.

<kPro>

This is the index of the first scan of this <subSeg> in the ideal full-profile. The first index is 0 (c code convention).

<numPro>

Is the total number of scans, over all <subSeg> in the profile. It can help determine if all of the <subSeg> packets have been received. Along with <kPro>, it can also help determine if the collected profile made it to the surface, or stopped before (say the ascent timed-out).

Ideal Full Profile

Each sensor has up to five depth regimes, where each regime has different sampling intervals. Together they define the full profile sampling scheme from 0-2000m (full depth). Given these parameters, the "ideal full profile" depth values can be calculated. For instance, say for pH:

```
z_ph = [ 2:2:1000 1050:50:2000 ];          %matlab format
z_ph = sort( z_ph, 'descend' );            %to sort high to low
```

Represents the desired sample depths for the ideal full profile, lowest index = deepest scan.

However, the real profile might start at 1900m. Using c code convention (0=first index), then <kPro> = 2 for <subSeg>=0 signifies scans at 2000m, 1950m were not taken.

GPS Packet

<sensor_ID> = 0x00 = "sen_NAV" in pkt_def.h = GPS sensor family. In this family are:
<data_ID> =

```
navID_gpsStartMiss=0x0, // 0 = start-mission GPS
navID_gpsStartDive,     // 1 = start-dive GPS
navID_gpsEndDive,       // 2 = end-dive GPS
navID_gpsAbort,         // 3 = abort GPS
navID_gpsBistOK,        // 5 = GPS Built-In-Self-Test (passed)
navID_gpsBistFail,      // 6 = GPS Built-In-Self-Test (failed)
```

All four <data_ID> are packed with the same format.

Let Dat[ ] = hexadecimal character array of the sensor packet's data (after the header).
Let Dat(j, n) = n-character value starting at Dat index j; First char at j=0.
UNLESS stated otherwise, all values are integers.

```
Dat( 0, 2)  valid if GPS is not good, valid=0, else ==-2 if West, +2 if East.
Dat( 2, 8)  kLat  signed latitude  degrees * 1e7 .
Dat(10, 8)  kLon  signed longitude degrees * 1e7.
Dat(18, 4)  week  GPS week.
Dat(22, 2)  wDay  week-day.
Dat(24, 2)  hour  UTC hour.
Dat(26, 2)  mins  UTC minute
Dat(28, 2)  tFix  time [10s per count] to get the fix.
Dat(30, 2)  nSat  Number of satellites used to compute the fix.
Dat(32, 2)  sMin  minimum SNR for the constellation used.
Dat(34, 2)  sAvg  average SNR.
Dat(36, 2)  sMax  maximum SNR.
Dat(38, 2)  HDOP  10*(Horizontal dilution of precision
Dat(40, 2)  ';'   end-of-packet character.
```

Engineering Packets

<sensor_ID> = sen_EngDat = 0x04 = engineering data family. The <data_ID> options are:

Value	Name	Description
0x00	engID_flt	rise and fall time-series
0x02	engID_sci	engineering parameters for the BGC Science board
0x04	engID_hyd_li	hydraulic pump time-series
0x0A	engID_diag	diagnostics of the first dive
0x0B	engID_prof	engineering params during profile-mode
0x0C	engID_beacon	eng. in beacon-mode
0x0D	engID_BITok	diagnostics of a good Built-In-Test
0x0E	engID_BITbad	diagnostics of a bad Built-In-Test

The software version of the engineering packet is stored in byte#5 of the sensor header:

Eng_ver = <proType>*16 + <proDir>

Eng_ver is used to identify future software variations and what variables are included.

For the following, let k=0 for the first variable, and assume for each entry that k is incremented by the size of the last variable (book-keeping of k is done in the processing code).

Exception Status

The exception status, **exc_stat**, is a 2-byte value with bit-status flags for each bit. It is the same for all engineering packets:

exc_stat exception status bits

```
0x0001 Valve failed to open
0x0002 Valve failed to close
0x0004 Questionable pressure
0x0008 Antenna was toggled
0x0010 Antenna switch failure. (no satellites even after toggling)
0x0020 GPS communication error (cannot talk to GPS unit)
0x0040 SBE binning errors
0x0080 Float took too long to leave the surface. (toggled valve)
0x0100 Had to restart the SBE in profile mode.
0x0200 Too many bad pressure values.
0x0400 Fell too deep during the drift phase.
0x1000 Valve failure during Sink phase of mission
0x2000 Valve failure during the Ascent phase of the mission.
```

abortFlag abort flag values

```
NoError        = 0,
BAD_P          = 2,          too many bad pressures.
BAD_XMITS      = 3,          >10 dives with no transmissions
SHORE_CMD      = 4,          received a shore command to abort
FINAL_DIVE     = 5,          reached the max #dives for this mission.
BAD_PHASE      = 6,          not in either beacon or profile mode
CANT_SURFACE   = 7,          Ascend() pumped all but P>3 dBar.
BAD_BEACON_XMITS  = 8,       >48 abort cycles with no good satellite xmits.
TIMED_OUT_WAITING = 9,       timed-out waiting to be deployed
WD_TIMER       = 32          reset due to watch-dog time-out.
```

Engineering Flight (data_ID = 0x00)

The rise and fall data are sent with this <data_ID>.
<proDir> = 0 if it is rise data, =1 if it is fall data.
Otherwise the data processing is the same as the SOLO-II.

Engineering Pump (data_ID = 0x04)

This is the list of the associated statistics with each hydraulic pump.
The data processing is the same as the SOLO-II.
However, the pump, fall and rise time-series have a modified list of the dive-phase values:

```
DIVE_PHASE_START              = 0,        // dive starting
DIVE_PHASE_START_OF_SINK      = 1,
DIVE_PHASE_PUMP2TARGET        = 2,
DIVE_PHASE_SEEKING            = 3,    // slowing down descent to neutral
DIVE_PHASE_BEGINNING_OF_DRIFT = 4,
DIVE_PHASE_SEEK_DURING_DRIFT  = 5,
DIVE_PHASE_BEGINNING_OF_FALL_TO_PROFILE =6 ,
DIVE_PHASE_START_OF_PREASCEND = 7,
DIVE_PHASE_START_OF_RISE      = 8,
DIVE_PHASE_END_OF_RISE        = 9,
DIVE_PHASE_ICE_TURNAROUND     =10,
DIVE_PHASE_DESCEND            =11,  // sent when just leaving the surface
DIVE_PHASE_DRIFTING           =12,  // reached the target depth and drifing
DIVE_PHASE_DESCEND_TO_PROFILE =13,
DIVE_PHASE_ASCEND             =14,  // ascending
DIVE_PHASE_SURFACE            =15, // reached the surface
```

Engineering BGC Science(data_ID = 0x02) DRAFT VERSION

This engineering packet has parameters specific to the BGC Science board.
The “Eng_Ver” is located in the sensor header, nibble <subSeg>

Dat(0, 4)	nWritten	#of files written to the SD card for this profile
Dat(k, 4)	mbFree	#of mB free on the SD card
Dat(k, 4)	ctdParseErr	Number of errors trying to parse the CTD PTS while profiling.
Code >=2.4	mar22	
Dat(k, 4)	jumpCtr	# of erroneous pressure jumps since profile start mar22
Dat(k, 2)	‘;’	end-packet termination character

Engineering Diagnostic-Dive (data_ID = 0x0A)

The following is the same as SOLO-II, except for the addition of the pH bias battery at the end, and not starting with the “Eng_ver” which is now sent in the sensor header packet.

Dat(0, 4)	nQueued	# of data blocks queued for this dive.
Dat(k, 4)	nTries	# tries to connect during the last surfacing.
Dat(k, 4)	parXstat	parse_X_reply status in the last surf session.
Dat(k, 4)	SBDIstat	ATSBD return status in last surface session

Dat(k, 4)	SBDsecs	Seconds taken in sending last SBD message
Dat(k, 4)	Vcpu	present CPU battery voltage counts 0.01V
Dat(k, 4)	Vpmp	present pump battery counts 0.01V
Dat(k, 4)	Vple	Pump battery counts at the end of last pump 0.01V.
Dat(k, 4)	vacBIT	pcase vacuum @BITest. 0.01inHg/cnt
Dat(k, 4)	vacBefore	pcase vacuum before filling oil bladder at surface 0.01 inHg
Dat(k, 4)	vacAfter	pcase vacuum after filling bladder at surface 0.01 inHg
Dat(k, 4)	ISRID	ID of the last interrupt.
Dat(k, 4)	HPavgI	avg pump I at bottom, LSB=1ma.
Dat(k, 4)	HPmaxI	max pump I at bottom, LSB=1ma.
Dat(k, 4)	PumpT_Hi	total time[s] pumped out, up to the last surface pump.
Dat(k, 4)	PumpT_Lo	Surface pump time [s]
Dat(k, 4)	SPRX	Surf press before resetoffset (pertains to prev dive).
Dat(k, 4)	SPRXL	press after resetoffset (pertains to prev dive).
Dat(k, 6)	diagP0	P0, T0, S0 = at water-start detect
Dat(k, 6)	diagT0	
Dat(k, 6)	diagS0	
Dat(k, 4)	SBnScan	# scans recorded by SBE: **SIGNED VALUE** -1 = could not get a scan count from the SBE, -2 = SBE never started the profile.
Dat(k, 4)	SBstat	SBE status SBntry&0xf)<<4) | ((DP->SBstrt&0x3)<<2) | (DP->SBstop&0x3);
Dat(k, 6)	diagP1	P1, T1, S1 = scan at beginning of ascent.
Dat(k, 6)	diagT1	
Dat(k, 6)	diagS1	
Dat(k, 4)	vacBIT	BITest vacuum 0.01 inHg
Dat(k, 4)	ampBIT	BIT motor current, 1 mA/cnt
Dat(k, 4)	secBIT	BIT motor run-time, 1 s/cnt
Dat(k, 4)	oilBIT	BIT oil counts
Dat(k, 4)	VpmpBIT	BIT pump voltage 0.01 V/cnt
Dat(k, 4)	VcpuBIT	BIT cpu voltage 0.01 V/cnt
Dat(k, 4)	exc_stat	exception status bits, see above
Dat(k, 2)	ventTm	0.1 secs that the air vent motor ran.
Dat(k, 2)	ventStat	LLD status before and after the air vent ran.
Dat(k, 4)	abortFlag	abort status flag,, see above
Dat(k, 2)	CPUtemp	temperature of the CPU: degC = CPUtemp/8.0 -2.0;
Dat(k, 2)	RH	relative humidity [%]
NEW TO BGC		
Dat(k, 4)	phBat0	pH bias battery right after turning off (end-of-surface)
Dat(k, 2)	';'	sensor-packet termination character.

Engineering Profile-Mode (data_ID = 0x0B)

The following is the same as SOLO-II, except for the addition of the pH bias battery at the end, and not starting with the “Eng_ver” which is now sent in the sensor header packet.

Dat(0, 4)	nQueued	# of data blocks queued for this dive.
Dat(k, 4)	nTries	# tries to get the data block receive status.
sn01:		nTries = (nDropped)*100 + nReq
		nDropped = satellite dropped out during the request.
		nReq = # of requests of the receive status.
>sn02		MSB = errByte of the Iridium session: 0=no error,
		-5 = Data-Carrier-Drop (lost connection).
		LSB = nTries calling into the server
Dat(k, 4)	parXstat	parse_X_reply status in the last surf session.
Dat(k, 4)	SBDIstat	ATSBD return status in last surface session
Dat(k, 4)	SBDsecs	Seconds taken in sending last SBD message
Dat(k, 4)	Vcpu	present CPU battery voltage counts 0.01V
Dat(k, 4)	Vpmp	present pump battery counts 0.01V
Dat(k, 4)	Vple	Pump battery counts at end of last pump 0.01V.
Dat(k, 4)	vac50m	pcase vacuum @50m descent. 0.01inHg/cnt
Dat(k, 4)	vacBefore	pcase vacuum before filling oil bladder at surface 0.01 inHg
Dat(k, 4)	vacAfter	pcase vacuum after filling bladder at surface 0.01 inHg
Dat(k, 4)	ISRID	ID of the last interrupt.
Dat(k, 4)	HPavgI	avg pump I at bottom, LSB=1ma.
Dat(k, 4)	HPmaxI	max pump I at bottom, LSB=1ma.
Dat(k, 4)	PumpT_Hi	total time[s] pumped out, up to the last surface pump.
Dat(k, 4)	PumpT_Lo	Surface pump time [s]
Dat(k, 4)	SPRX	Surf press before resetoffset (pertains to prev dive).
Dat(k, 4)	SPRXL	press after resetoffset (pertains to prev dive).
Dat(k, 6)	diagP0	P0, T0, S0 = scan at the start of ascent.
Dat(k, 6)	diagT0	
Dat(k, 6)	diagS0	
Dat(k, 6)	diagP1	P1, T1, S1 = shallowest scan in profile.
Dat(k, 6)	diagT1	
Dat(k, 6)	diagS1	
Dat(k, 4)	nBad	# of bad SBE bins.
Dat(k, 4)	SBnScan	# scans recorded by SBE: **SIGNED VALUE**
		-1 = could not get a scan count from the SBE,
		-2 = SBE never started the profile.
Dat(k, 4)	SBstat	SBE status
		SBntry&0xf)<<4) | ((DP->SBstrt&0x3)<<2) | (DP->SBstop&0x3);
Dat(k, 6)	DPavg0	avg P over the first half of the drift phase
Dat(k, 6)	DTavg0	avg T over the first half of the drift phase.
Dat(k, 6)	DSavg0	avg S over the first half of the drift phase.
Dat(k, 6)	DPavg1	avg P over the 2nd half of the drift phase

Dat(k, 6)	DTavg1	avg T over the 2nd half of the drift phase.
Dat(k, 6)	DSavg1	avg S over the 2nd half of the drift phase.
Dat(k, 4)	fallTm	time [s] from start-fall to 50m.
Dat(k, 4)	fallW	descent speed [mm/s] over the top 50m.
Dat(k, 4)	seekT	time [s] of pumping in 1st seek.
Dat(k, 4)	seekP	0.1 dbar change in 1st seek. **SIGNED VALUE**
Dat(k, 4)	exc_stat	exception status bits, see above
Dat(k, 2)	ventTm	0.1 secs that the air vent motor ran.
Dat(k, 2)	ventStat	LLD status before and after the air vent ran.
Dat(k, 4)	pOffset	pOffset * 800 = SBE pressure offset. **SIGNED VALUE**
Dat(k, 4)	seekSc	# of 0.1 s ticks pumped to the target depth.
Dat(k, 4)	nSent	# of Iridium messages sent during the last surfacing.
Dat(k, 2)	iceStat	ice status (1=on, 0=off, 2=tripped, 3=thaw_delay).
Dat(k, 1)	binMod	current CTD binning mode (stored in one nibble).
Dat(k, 1)	subCyc	current sub-cycle (0, 1 or 2: stored in one nibble).
Dat(k, 2)	CPUtemp	temperature of the CPU: degC = CPUtemp/8.0 -2.0;
Dat(k, 2)	RH	relative humidity [%]
NEW TO BGC		
Dat(k, 4)	phBat0	pH bias battery right after turning off (end-of-surface)
Dat(k, 4)	phBat1	pH bias battery right before turning on (mid-drift).
Dat(k, 2)	';'	end-sensor packet termination character.

Engineering Beacon (data_ID = 0x0C) `engID_beacon`

This is the engineering packet when the float is beacon-mode, also called “abort”.
There is no difference between the BGC float and the standard SOLO-II float.

Dat(0, 4)	nQueued	# of data blocks queued for this dive.
Dat(k, 4)	nTries	# tries to connect during last surfacing.
Dat(k, 4)	parXstat	parse_X_reply status in the last surf session.
Dat(k, 4)	SBDIstat	ATSBD return status in last surface session
Dat(k, 4)	SBDsecs	Seconds taken in sending last SBD message
Dat(k, 4)	Vcpu	present CPU battery voltage counts 0.01V
Dat(k, 4)	Vpmp	present pump battery counts 0.01V
Dat(k, 4)	vacNow	pcase vacuum @start of this abort-cycle transmit.
Dat(k, 4)	vacAbrt	pcase vacuum @entering the abort cycle.
Dat(k, 4)	ISRID	ID of the last interrupt.
Dat(k, 4)	abortFlag	Abort flag that caused the abort, see link here.
Dat(k, 2)	CPUtemp	temperature of the CPU: degC = CPUtemp/8.0 -2.0;
Dat(k, 2)	RH	relative humidity [%]
Dat(k, 2)	‘;’	end-sensor packet termination character.

Engineering BIT-OK (data_ID = 0x0D) `engID_BITok`

Engineering Bit-Fail (data_ID = 0x0E) `engID_BITbad`

The Built-In-Test either passes (OK) or fails, triggering two different <data_ID>s. The only difference is the Bit-Fail writes out the **bitStatus** field, whereas BIT-OK does not.
There is no difference between the BGC float and the standard SOLO-II float.

Dat(0, 4) nQueued # of data blocks queued for this dive.
=== bitStatus is only present if data_ID == Bit-Fail ========================
Dat(k, 4) bitStatus BIT status bit-flags: “quickBIT” is fast test, “full BIT” is a full test.

```
BB_RAM        0x0001 RAM bad (not used).
BB_ROM,       0x0002 ROM bad (not used).
BB_EEPROM,    0x0004 EEPROM test failed (quickBIT).
BB_VENT,      0x0008 Air vent failed.
BB_VCPU,      0x0010 CPU Voltage low  (quickBIT).
BB_VPUMP,     0x0020 Pump Voltage low (full and quickBIT).
BB_LOVAC,     0x0040 Low Vacuum  (quickBIT).
BB_SLOPUMP,    0x0080 slow oil pump  (MxHiP sec exceeded) (full BIT).
BB_PUMPCUR,    0x0100 PumpCur too high or low (full BIT).
BB_SCI,       0x0200 SBE comm fail (full BIT).
BB_VALVE,     0x0400 Valve open or close failure (full BIT).
BB_HIVAC,     0x0800 Air vacuum high (>1100) (full BIT).
BB_IRIDIUM,    0x1000 Comms failed to iridium modem (full BIT).
BB_GPS,       0x2000 Comms failed to gps device (full BIT).
BB_OILVAC,    0x4000 HP pump didn't run long enough (full BIT).
BB_PH_BIAS     0x8000 pH Bias Battery too low (quickBIT).\
```

=== the rest of the fields are shared with both <data_ID>s. ===============

Dat(k, 4)	pOffset	pOffset * 800 = SBE pressure offset. **SIGNED VALUE**
Dat(k, 4)	Vcpu	present CPU battery voltage counts 0.01V
Dat(k, 4)	Vpmp	present (no-load) pump battery counts 0.01V
Dat(k, 4)	Vple	Pump battery counts at end of last pump 0.01V.
Dat(k, 4)	HPavgI	avg pump I, LSB=1ma.
Dat(k, 4)	PumpT_Lo	BIT pump time [s]
Dat(k, 2)	oilB4	oil counts before pumping.
Dat(k, 2)	oilAfter	oil counts after pumping.
Dat(k, 4)	vacBefore	pcase vacuum before pumping, 0.01 inHg
Dat(k, 4)	vacAfter	pcase vacuum after pumping, 0.01 inHg
Dat(k, 2)	openN	#of tries needed to open the valve.
Dat(k, 2)	closeN	#of tries needed to close the valve.
Dat(k, 4)	ISRID	ID of the last interrupt.
Dat(k,60)	sbeStr	30 byte string from the SBE reply to “PTS”
Dat(k, 2)	CPUtemp	temperature of the CPU: degC = CPUtemp/8.0 -2.0;
Dat(k, 2)	RH	relative humidity [%]
Dat(k, 2)	‘;’	sensor-packet termination character.

BGC Parameter Packet

Let **NSENSORS** = 6 = total #sensors; sensorNames = { ctd, dox, ph, eco, ocr, no3 }

The BGC-SOLO has two output packets for the named eeprom parameters: the normal SOLO-II and those associated with the BGC Science board, just referred to as “SCI” here.

The SCI parameters can be split into two categories: eeprom-named values, much like SOLO-II; and arrays of un-named values.

sen_IRD (0x01) = **<Sensor_ID>** associated with Iridium products.

irdID_eeprom (0x03) = **<data_ID>** for the eeprom ascii dump.
Unpack the ascii string just like the normal SOLO-II eeprom dump.

irdID_psm (0x04) = **<data_ID>** for the SCI parameter arrays.
See below for unpacking.

irdID_psm

BGC SCI parameter arrays. All parameters below are signed two-byte (4 hex-char) values, with a range of -32768...32767.

Let Dat[ ] = hexadecimal character array of the sensor packet's data (after the header).
Let Dat(j, n) = n-character value starting at Dat index j;

```
Dat(0, 4) / 2    = nSenVar      =number of sensor variable entries
Dat(4, 4) / 2    = nRegVar      =number of sensor depth region entries
Dat(8, 4) / 2    = nCoefVar     =number of sensor coefficients
```

Sensor Parameters:

```
N = nSenVar/N_SENSORS; // = #of parameters for each sensor.  Should be = 5.
k = 12;  // = first index into Dat to start unpacking
For i=0; i<N_SENSORS;i++
        // we are unpacking sensor i; name = sensorNames[ i ];
        For j= 0; j<N; j++
                x= Dat(k, 4); k = k+4;  // read in 4hex = 2 bytes
                Val[i,j] =swapBytes(x);// value is sent as Little-Endian; linux wants Big-End.
                        Take care to get the correct signed value with the byte swap.
        End for j
End for i
```

Val[i, j] = jth parameter for the ith sensor.

```
j=0     <EnSen>       = {0, 1}: if ==1, sensor is enabled.  If==0, then is not used.
j=1     <EnDrift>     = % of drift samples to take data for this sensor (0-100).
j=2     <DecDrift>    = decimation for this sensor during the %drift time.
j=3     <dataT>       = dataType; can select a particular data-product .
j=4     <packT>       = packType (see above <packType>).
```

The last value of k from the above code is ready to unpack the next array: Depth regions

Region Parameters:

```
N = nRegVar/ N_SENSORS;
NR = N / MAX_REGIONS  (MAX_REGIONS = 5)    =#points for each region
For i=0; i<N_SENSORS;i++
        // we are unpacking sensor i; name = sensorNames[ i ];
        For j= 0; j<MAX_REGIONS ; j++
                // unpack for each depth region
                For kr = 0; kr<NR; kr++
                        // unpack for each param in this region
                        x= Dat(k, 4); k = k+4;  // read in 4hex = 2 bytes
                        Reg[i,j, kr] =swapBytes(x);     // Little-End to Big-Endian
                End for kr
        End for j
End for i
```

The four valid parameters for each depth region are:

kr=0	<zMin>	min depth for this region
kr=1	<zMax>	max depth for this region
kr=2	<dz>	depth interval between scans
kr=3	<scanT>	can define scan options, presently not used.

Coefficient Parameters:

N = nCoefVar/N_SENSORS = number of coefficients for each sensor. Should be = 2.
The last value of k from the above code is ready to unpack the next array: Depth regions

```
For i=0; i<N_SENSORS;i++
        // we are unpacking sensor i; name = sensorNames[ i ];
        For j= 0; j<N; j++
                x= Dat(k, 4); k = k+4;  // read in 4hex = 2 bytes
                Val[i,j] =swapBytes(x);// value is sent as Little-Endian; linux wants Big-End.
        End for j
End for i
```

Val[i, j] = jth coefficient for the ith sensor.

j=0	<offset>
j=1	<gain>

Let <counts> = profile array's unpacked integer value.

<sciValue> = (<counts> - off)/<gain>

NOTE some sensor variables have hard-coded offset and gain values. This will be covered later.

BGC Profile Packing Algorithm

Bad Scans

For all three algorithms (p1_Type = none, 1st-diff, 2nd-diff), the profile is broken into NSUB=16 scan chunks. Each chunk, no matter the algorithm, starts with.

<byte0> = **<bitBadFlag><numTaken_m1>**

Where

<bitBadFlag> = **<byte0>** & 0x80 (=last bit).

It is set if there are bad/missing scans within this NSUB.

<numTaken_m1> = **<byte0>** & 0x0f = LSNibble of **<byte0>**

Define **<numTaken>** = **<numTaken_m1>** + 1; = #of good+bad scans in this chunk

If **<numTaken_m1>** =0xf = 15: **<numTaken>**= 16.

=0x0 ; **<numTaken>** = 1.

Note nothing is packed if **<numTaken>**=0, so **<numTaken_m1>** is always well defined.

If **numTaken** < **NSUB**, then this chunk is a subset = end-of-profile subset.

If **<bitBadFlag>** is set, then **<byte0>** is followed by two more bytes.

<badBits> = **<byte1b><byte2b>** = 16 bits, each bit representing one scan.

<byte2b> is the LSB, it's LSbit = to the first scan, its MSbit to the eighth.

<byte1b> is the MSB, it's LSbit = 9th scan, MSbit to the 16th.

The bit is clear (0) if the scan is good, or not taken (happens with the end-of-profile subset).

The bit is set(1), if that scan is bad.

Let **<numBad>** = sum(each bit set(1) in badBits). Then

<numGood> = **<numTaken>** - **<numBad>**.

After **<badBits>**, there are **<numGood>** scans of data, packed as per **<packType>** .

The output array for this profile chunk will consist of **<numTaken>** scans, with the bad scans set in **<badBits>** set to NAN, and the others filled with the good values.

p1_Type = 0 (none):

The **<numGood>** data follows, each scan using (**<p2_Type>**+1) bytes.

AutoGain

If **p2_type = 2**, and **p1_Type = 1 or 2** (1st-diff or 2nd-diff) then the BGC shifts and rescales the NSUB scans so their range is 0..65535 (2-byte value), creating an **<autoOffset>** and **<autoGain>**. This is needed for the OCR downwelling irradiance and pH, which have a larger dynamic range. Both OCR and pH have their own auto-scaling algorithms, which will be discussed later. If the AutoGain is being used, the start of each NSUB segment begins with:

<byte0> = **<bitBadFlag><numScans_m1>**

(potentially 2 more bytes if **<bitBadFlag>** is set)

<byte1a> = **<autoOffset>**

<byte2a> = **<autoGain>**

The rest of the NSUB segment is the same as if the AutoGain option was not being used.

One unpacking technique is to handle both auto-gain and non-auto-gain:

1. Set autoOffset = 0, autoGain = 1 no matter the chosen option.
2. If it is auto-gain, read in the new values of autoOffset, autoGain.
3. Do the first-difference, second-difference unpacking, just as normal.
4. Apply the autoOffset, autoGain to the segment (if =0, 1, no changes are made).

The above requires the least code mods to handle the AutoGain option.

BGC First-Diff

Here is the header for each first-difference NSUB segment:

<byte0> = **<bitBadFlag><numScans_m1>**

(potentially 2 more bytes if **<bitBadFlag>** is set)

(potentially 2 more bytes if AutoScaled)

<byte1a> = **<autoOffset>**

<byte2a> = **<autoGain>**

<byte1> = **<gain>** to get the 1st-diff into one-byte values: +/-127.

<byte2><byte3> = value of the first good scan of NSUB.

Followed by (NSUB-1) bytes of the first-difference of the rest of the scans.

The first-difference data are unpacked with the same original SOLO algorithm.

BGC Second-Diff

The 2nd-difference algorithm calculates the number of nibbles needed to successfully pack the NSUB chunk, with the number of nibbles ranging from 1 to 6. The value sent back is the numNib -1 (range 0..5). This is stored in bits 4-6 of **<byte0>**

<bitBadFlag> = **<byte0>** & 0x80 (=last bit).

<nNib_m1> = (**<byte0>** & 0x70) >>4 (=0..7, only 0..5 is used)

<numDat_m1> = **<byte0>** & 0x0f

<numNib> = **<nNib_m1>** + 1; = number of nibbles for each 2nd-diff scan.

(potentially 2 more bytes if **<bitBadFlag>** is set)

(potentially 2 more bytes if AutoScaled)

<byte1a> = **<autoOffset>**

<byte2a> = **<autoGain>**

<byte1><byte2> = value of the first good point in NSUB.

<byte3><byte4> = value of the first-difference between the first two good points..

Followed by (NSUB-2) * **<numNib>** nibbles of the 2nd-difference of the rest of the scans.

If (NSUB-2)***<numNib>** is odd, the last nibble is written out as a byte (the NSUB end-of-segment always falls on a byte boundary). Except for the **<numDat_m1>** being part of each NSUB (instead of at the head of the full sensor packet) and each NSUB ending on a byte boundary, the second-difference unpacking algorithm is the same as the original SOLO.

AutoGain Specifics

The standard first-difference and 2nd-difference algorithms assume that the integer values are two bytes (0..65535). However, both the OCR and pH sensors have a larger dynamic range. The AutoGain BGC algorithm calculates an offset and gain for every NSUB chunk so that NSUB values are between 0..65535.

OCR AutoScale

The downwelling irradiance has a depth response of ~exp(- z/ lambda) where lambda is an 1/e absorption depth. The a/d counts output has a range of 0x01e000 < cnts < 0x140000, or 14x the dynamic range of two bytes. Its exponential response points to a gain that also responds exponentially. Let `y[k]` be the OCR time-series that has already been re-scaled by the nominal offset and gain values. It is the time-series that needs to be auto-scaled.

Offset: The offset is calculated by the BGC, assuming NSUB `y` values, with `ymin` = min value.

```
#define OCR_OFF_SCALAR 10000 // each OCR_off LSB = 10000 counts
      off  =  floor(  ymin /OCR_OFF_SCALAR  );
      if (off<0) off=0;  // 'off' is the byte value sent via Iridium.
      off0 = off*OCR_OFF_SCALAR; // 'off0' is the value to subtract from y
```

<autoOffset> = `off` .

Gain: `ymax` = max (`y` - `off0`) for the NSUB values.

```
#define Y2MAX   65534  // max counts range for unpacked unsigned 2-byte output
      p  = 0;  // for OCR the divider is 2^p
      gn = 1; // 2^0
      for (j=0; j<32; j++) // --------------
      {
      if ( ymax > Y2MAX )
      { ymax = ymax >> 1; // divide by 2
      p++; // increase exponent
      gn = gn << 1; // multiply by 2
      }
      else { break; } // all done!
      } // ---------------------------------
```

<autoGain> = `p; gn = 2^p;`

The BGC float re-scales `y` to `Y'` using:

```
      Y' = ( y - off0 )/ gn;
```

And then `Y'` is packed using the first or second-difference algorithm.

To convert the returned RUDICS counts, `Y'`, to the OCR BGC counts, apply:

gn = 2^**<autoGain>**

off0 = **<AutoOffset>***`OCR_OFF_SCALAR`

```
y  = Y' * gn + off0
```

pH AutoScale

The pH output can be from -2.5 to +2.5 V at a 1 uV resolution, equivalent to 5e6 range. However the typical range for a profile is 0.1V = 100,000 uV, slightly greater than the 65536 2^16 range. The biggest problem with pH is drift, so the dynamic offset will keep values in range.

Offset: The offset is calculated by the BGC, assuming NSUB y values, with ymin = min value.

```
#define  PH_OFF_SCALAR 20000 // each  PH_off LSB = 20000 uV = 0.02V
      off  =  floor(  ymin /PH_OFF_SCALAR  );
      if (off<0) off=0;  // 'off' is the byte value to transmit.
      off0 = off*PH_OFF_SCALAR; // 'off0' is the value to subtract from y
```

<autoOffset> = `off .`

Gain: ymax = max (y - off0) for the NSUB values.

```
#define Y2MAX   65534  // max counts range for unpacked unsigned 2-byte output
      gn = ( ymax + Y2MAX/2 ) / Y2MAX;  // scalar, rounded up
      if (gn>255) gn= 255;   // limit to its max value
      if (gn<  1) gn=   1;   // limit to its min value
```

<autoGain> = `gn;`

The re-scaled values of y are calculated:

```
      Y' = (y - off0 )/ gn;
```

And then Y' is packed using the first or second-difference algorithm.

To convert the returned RUDICS counts to the PH microvolts, apply:

gn = **<autoGain>**

off0 = **<AutoOffset>***`OCR_OFF_SCALAR`

```
PH_uv = rudics_counts * gn + off0 - 2,500,000
```

Nitrate Spectra

One scan consists of ~40 spectral bins, S[k]. The number of bins depends upon each SUNA, but will always be <50, and most likely in the 40-42 range.

The <data_ID> for S[k] is set to no3ID_sk = 0x0f. The index k is sent in byte #5 of the sensor header, normally equal to <proType> and <proDir>. Since no3 is just sampled on ascent at set depth intervals, these two values are used for k: k = <proType>*16 + <proDir>

Profile Counts to EngUnits

After the profile is unpacked to count values it must be rescaled to engineering units.

CTD

<sensor_ID> = 0x21 = 33 = CTD. The CTD data channels are
<data_ID> =

0 = CTD pressure.
1 = CTD temperature.
2 = CTD salinity.

The CTD data is packed by the main SOLO board, and is re-scaled just like SOLO-ii. The nominal offset and gain values plus equations are

P [dBar] = cnts / pGain - pOff; pOff = 10; pGain = 25;
T [degC] = cnts /tGain - tOff; tOff = 5; tGain = 1000;
S [PSU] = cnts/sGain - sOff; sOff = 2; sGain = 1000;

But these nominal values can also be changed via shore commands.
The latest values should be read in from the ‘A ‘ = argo information line, as S is usually changed to;

S [PSU] = cnts/sGain - sOff; sOff = -25; sGain = 5000;

BGC Scaling

To allow larger range of offset besides just integer scientific units, the BGC sensors switch to the following format

EngUnit = (cnts - senOff) / senGain; (cnts = EngUnit*senGain + senOff)

DO

<sensor_ID> = 0x23 = 35 = Dissolved Oxygen. The DO data channels are
<data_ID> =

0	= DO pressure	
1	= rawPhase	= time delay [us]
2	= thermV	= thermistor volts
3	= DO	= calculated DO by the SBE83
4	= thermT	= calculated thermistor T

By setting the sensor variable <dataT> the user can choose to send back
Default <dataT> = 0: send channels 1 (rawPhase) and 4 (thermT).
<dataT> = 1; send channels 1 (rawPhase), 3 (DO) and 4 (thermT).
<dataT> = 2; send all four; 1-4;

All of the DO variables use the same scalar as defined in the coefficient parameter list (thus shore-settable as well). The nominal values are:
EngDOX[k] = (cnts[k] - doxOff) / doxGain; doxOff = 5000; doxGain = 1000;

DO In-Air

When pressure is less than 2 dBar on ascent, the float's science board acquires the last BGC readings, packs the data, and begins the "in-air" DO time-series. This should occur ~5-10s after the "<2 dBar" trigger; so it may or may not still be in-water, depending upon the ascent speed.

This time-series is defined by two shore-settable parameters:

`sDOairTm_s`	= seconds between DO samples (typically 15s).
`sDOairN`	= number of samples to acquire.

The DO in-air time series is uniquely marked by setting <proType>=2 (==raw data), <proDir> = 2 (==ascent, discrete) and <reserved> = 1.

The DO in-air pressure time series (<data_ID>=0) has a valid reading for the first sample(s). However, when the SOLO main CPU logic decides it is on the surface (could be up to ~60s delay), it takes control over the CTD to pack the profile data, and the pressure sensor is no longer available. The BGC science board packs the "in-air" pressure time-series data using the last valid reading. Except for the first reading, the "in-air" pressure series is not valid (will simply repeat the last valid reading). Therefore the pressure should not be used to decide if the DO is submerged or not.

pH

<sensor_ID> = =0x24 = 36 = pH. The pH data channels are
<data_ID> =

0 = pH pressure

The pH sensor outputs four variables:

1 = <vRef> = reference electrode voltage.
2 = <Vk> = counter electrode voltage.
3 = <Ik> = counter electrode current.
4 = <Ib> = ISFET body current.

<Vref> and <Vk> are in volts, <Ik> and <Ib> are in nano-amps.
For <Vref> and <Vk> the offset and gain are fixed for 1 uV resolution and V shifted by 2.5 V (one count = 1 uV)

pH[1,2] [Volts] = (cnts[k[- vOff)/vGain; vOff = 2.5e6; vGain = 1e6;

For <Ik> and <Ib> (counts = 0.02 nA resolution)

pH[3,4] [nA]= (cnts[k] - phOff) / phGain; phOff = 25000; phGain = 50;

But the pH offset and gain of <Ik>, <Ib> can be set by shore, so read it in from the coefficients list.

ECO

<sensor_ID> = 0x22 = 34 = ecoTriplet. The data channels are
<data_ID> =

0 = ECO pressure

The EcoTriplet outputs three variables:

1 = Chlorophyll Fluorescence.
2 = Backscatter Turbidity
3 = CDOM.

All three outputs are in counts 0..4100 range.
All of the ECO variables use the same scalar as defined in the coefficient parameter list (thus shore-settable as well). The nominal values are:

ECO[k] = (cnts[k] - ecoOff) / ecoGain; ecoOff = 500; ecoGain = 1;

OCR

<sensor_ID> = 0x25 = 37 = OCR504 . The data channels are
<data_ID> =

0 = OCR pressure

The OCR504 outputs four variables:

1 = 380 nm
2 = 412 nm
3 = 490 nm
4 = PAR.

The OCR is configured to output counts, which will require the SBE calibration coefficients to convert to engineering units.

Let OCR_full[k] = full counts for channel k as output by the OCR. The first step performed by the BGC float is to divide by OCR_COUNTS_DIV = 1024 and shift down by OCR_COUNTS_OFF = 0x1e,000. Let OCR_count0[k] be these new values.

```
#define  OCR_COUNTS_DIV 1024  // first divide OCR counts by 1024
#define  OCR_COUNTS_OFF 0x1e0000 // and then subtract the offset  = 1966080

      OCR_count0[k] =  OCR_full[k]/ OCR_COUNTS_DIV  - OCR_COUNTS_OFF;
```

`OCR_count0[k]` is now in 0..1,310,720 range.

There are still offset and gain parameters set in eeprom, allowing the user to re-scale as well, but most likely they will remain at their default values of `OCR_OFF` = 0; `OCR_GAIN`= 1; The float applies these:

```
      y[k] =  OCR_count0[k]*OCR_GAIN + OCR_OFF ;
```

The offset and gain values should be read in from the PSM coefficient parameters. The `y[k]` time series is next auto-scaled.

Finally, the float applies the autoscaling to every 16-scan chunk to `y[k]` (<packType> = 6), applying a dynamic offset and gain: see autoScale OCR section. This creates a Y' time-series, which is what is sent via Iridium. Follow the autoScale OCR section to convert back to `y[k]`. This applies to all four OCR channels. Given `y[k]`, convert to `OCR_count0[k]`

```
      OCR_count0[k] = ( y[k] - OCR_OFF ) / OCR_GAIN
```

And then back to the full OCR counts:

```
      OCR_full[k] = ( OCR_count0[k] + OCR_COUNTS_OFF ) * OCR_COUNTS_DIV
```

To convert `OCR_full[k]` counts to irradiance, apply the calibration coefficients for each channel. Each one has three SBE coefficients: `A0`, `A1`, and `im`. (im==immersed)

```
      OCR_eng[k]  = A1 * im * ( OCR_full[k] - A0 )
```

NO3

<sensor_ID> = 0x31 = 49 = SUNA Nitrate Sensor. The data channels are
<data_ID> =

0	=	NO3 pressure	

The BGC-SOLO can send up to five NO3 variables:

1	=	Res	= residual rms of the no3 calculation.
2	=	darkM	= mean of the dark spectrum.
3	=	no3	= nitrate as calculated by the SUNA.
4	=	SeaDark	= mean of the tail of the spectrum.
5	=	darkRMS	= RMS of the darkM calculation.

(The SUNA outputs 25 variables plus the spectrum. Eleven variables are stored in RAM, plus the spectra. Of those 11, the five above are deemed worth sending)

If <dataT>= 3, the user chooses to send all 5; <dataT>=2 ==send first 4;
<dataT>= 1 == send first 3; <dataT>=0 == send the first 2 == **DEFAULT**

<Res>	= (cnts - off0)/gn0;	off0 = 1000;	gain0 = 10000; **NOMINAL**
<darkM>	= (cnts - off1)/gn1;	off1 = 0;	gain1 = 1;
<no3>	= (cnts - off2)/gn2;	off2 = 100;	gain2 = 100;
<SeaDark>	= (cnts - off1)/gn1;	off3 = 0;	gain3 = 1;
<DarkRMS>	= (cnts - off1)/gn1;	off4 = 0;	gain4 = 1;

The eeprom calibration coefficients apply to <Res>, allowing the user to change its values via shore command.

Nitrate Spectrum
<data_ID> = 0x0f = 15.

The spectrum is sent back in counts (off=0, gain=1). More information can be found in its section above.

BIST Unpacking

BIST = Built-In-Self-Test.
The BIST is run for each sensor

a) Dockside check-out (will return with dive = -1).
b) Before start-ascent (deepest value). However, the status and error counter are from the end of the profile, displaying all error flags encountered and a total error count.

Hexadecimal data definition:

Let Dat[ ] = hexadecimal character array of the sensor packet's data (after the header).
Let Dat(j, n) = n-character value starting at Dat index j; First char at j=0.
UNLESS stated otherwise, all values are integers.

BIST does store some variables as floating-point values. Example c code for the conversion can be found here.

Fortran code can use the "equivalence" statement to equate the memory addresses of a float and an int. As with the c code, this may require some byte-swapping to get them in the correct order for your specific processor.

sen_BIST = 0x05 = BIST sensor family <sensor_ID>

BIST <data_ID> types:

```
bistIDcpu  =0x0,  // 0 main cpu BIST (NOT IMPLEMENTED)
bistIDsci,        // 1 science board BIST (NOT IMPLEMENTED)
bistIDctd,        // 2 SeaBird ctd
bistIDdox,        // 3 Dissolved Oxygen
bistIDph ,        // 4 pH sensor
bistIDeco,        // 5 ECO-triplet
bistIDocr,        // 6 OCR downwelling irradiance
bistIDno3a,       // 7 nitrate eng. subset
bistIDno3b,       // 8 nitrate spectrum dump
bistIDno3c,       // 9 nitrate ascii string
```

CTD BIST `bistIDctd = 2`

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.
Dat(8,4)	volt_cnt	Volts(counts); volt_cnt*0.01 = Volts.
Dat(12,4)	p_cnt	press(counts); dBar = p_cnt/25 - 10;
Dat(16,4)	ma0	milli-amps(ma) idle current (not sampling)
Dat(20,4)	ma1	ma with the CTD pump set to high-speed.
Dat(24,4)	ma2	ma with the CTD pump set to low-speed.
Dat(28,4)	ma3	ma with the CTD in sleep-mode.
Dat(32,1)	';'	end-of-sensor packet.

Status bit flags:

```
SBE_ok         =      0, // no error
SBE_Ack_Err    = 0x0001, // error getting an ack.
SBE_Start_Err  = 0x0002, // error with startingProfile
SBE_Stop_Err   = 0x0004, // error stopping the profile
SBE_Init_Err   = 0x0008, // error with init_params()
SBE_Pro_Err    = 0x0010, // error getting P,T,S during the profile
SBE_PTS_Err    = 0x0020, // error with getting the PTS sample
SBE_FP_Err     = 0x0040, // error with getting an FP sample.
SBE_busy       = 0x0080, // not done with last sample yet
SBE_parse      = 0x0100, // did not parse the command correctly
SBE_emu        = 0x1000, // emulation mode (either total or just press)
SBE_fatal      = 0x2000, // some fatal thing
SBE_soloctrl   = 0x4000, // solo2 has control of the sbe!!
```

DO BIST `bistIDdox = 3`

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.
Dat(8,4)	volt_cnt	Volts(counts); volt_cnt*0.01 = Volts.
Dat(12,4)	maMax	milli-amps(ma) max current while sampling.
Dat(16,4)	maAvg	average current (ma) during the sample.
Dat(20,8)	doPh	raw phase delay [us]: sent as a 4-byte float
Dat(28,8)	thmV	thermistor volts: 4-byte float.
Dat(36,8)	DO	calculated DO [mL/L]: 4-byte float.
Dat(44,8)	thmC	thermistor calculated degC: 4-byte float.
Dat(52,2)	';'	end-of-sensor packet.

Status bit flags:

```
dox_ok      =      0, // no error
dox_parse   = 0x0001, // error parsing the return string
dox_multi   = 0x0002, // failed all tries to parse
dox_tmo     = 0x0004, // timed-out from inactivity
dox_busy    = 0x0008, // not correct state when asked for sample
```

```
dox_badStart= 0x0010, // bad start of string to parse
dox_emu     = 0x0080, // emulation-mode
```

pH BIST `bistIDph =4`

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.
Dat(8,4)	volt_cnt	Volts(counts); volt_cnt*0.01 = Volts.
Dat(12,4)	maMax	milli-amps(ma) max current while sampling.
Dat(16,4)	maAvg	average current (ma) during the sample.
Dat(20,8)	vRef	reference electrode V: sent as a 4-byte float
Dat(28,8)	Vk	current electrode V: 4-byte float.
Dat(36,8)	Ik	counter electrode current [nA]: 4-byte float.
Dat(44,8)	Ib	ISFET body current [nA]: 4-byte float.
Dat(52,2)	';'	end-of-sensor packet.

Status bit flags:

```
ph_ok       =      0, // no error
ph_parse    = 0x0001, // error parsing the return string
ph_multi    = 0x0002, // failed all tries to parse
ph_tmo      = 0x0004, // timed-out from inactivity
ph_busy     = 0x0008, // not correct state when asked for sample
ph_badStart = 0x0010, // bad start of string to parse
ph_emu      = 0x0080, // emulation-mode
```

ECO BIST `bistIDeco = 5`

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.
Dat(8,4)	volt_cnt	Volts(counts); volt_cnt*0.01 = Volts.
Dat(12,4)	maMax	milli-amps(ma) max current while sampling.
Dat(16,4)	maAvg	average current (ma) during the sample.
Dat(20,4)	Chl	chlorophyll fluorescence counts (integer).
Dat(24,4)	bb	backscatter signal counts (integer)
Dat(28,4)	CDOM	CDOMfluorescence counts (integer).
Dat(32,1)	';'	end-of-sensor packet.

Status bit flags:

```
eco_ok      =      0, // no error
eco_parse   = 0x0001, // error parsing the return string
eco_multi   = 0x0002, // failed all tries to parse
eco_tmo     = 0x0004, // timed-out from inactivity
eco_busy    = 0x0008, // not correct state when asked for sample
```

```
eco_badStart= 0x0010, // bad start of string to parse
eco_emu     = 0x0080, // emulation-mode
```

OCR BIST `bistIDocr = 6`

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.
Dat(8,4)	volt_cnt	Volts(counts); volt_cnt*0.01 = Volts.
Dat(12,4)	maMax	milli-amps(ma) max current while sampling.
Dat(16,4)	maAvg	average current (ma) during the sample.
Dat(20,8)	ch01	channel 1 = 380 nm: counts (integer)
Dat(28,8)	ch02	channel 2 = 412 nm: counts (integer).
Dat(36,8)	ch03	channel 3 = 490nm: counts (integer).
Dat(44,8)	ch04	channel 4 = PAR: counts (integer).
Dat(52,2)	';'	end-of-sensor packet.

Status bit flags:

```
ocr_ok      =      0, // no error
ocr_parse   = 0x0001, // error parsing the return string
ocr_multi   = 0x0002, // failed all tries to parse
ocr_tmo     = 0x0004, // timed-out from inactivity
ocr_busy    = 0x0008, // not correct state when asked for sample
ocr_badStart= 0x0010, // bad start of string to parse
ocr_emu     = 0x0080, // emulation-mode
```

NO3 BIST

Nitrate has three BIST sensor packets:

```
bistIDno3a,        // 7 nitrate eng. subset
bistIDno3b,        // 8 nitrate spectrum dump
bistIDno3c,        // 9 nitrate ascii string
```

bistIDno3a = 7 eng. data

This has the engineering header parameters.

Dat(0,4)	status	status bit-flag. Values given below.
Dat(4,4)	numErr	number of errors during the profile.

Let **numVar** = SenHdr.proDir = <proDir> in the sensor packet (should == 11)
The next **numVar** 4-byte (8-char) values are floating-point values of:

```
J_err     =    0,  // 00 error counter
J_rh            ,  // 01 relative humidity
J_volt          ,  // 02 measured voltage
J_amps          ,  // 03 measured amps
J_darkM         ,  // 04 dark spectrum mean
J_darkS         ,  // 05 dark spectrum stdDev
J_no3           ,  // 06 nitrate
J_res           ,  // 07 residual RMS
J_fit1          ,  // 08 FIT Pixel Begin
J_fit2          ,  // 09 FIT Pixel End
J_Spectra       ,  // 10 first spectrum channel value
J_SeaDark       ,  // 11 Seawater Dark, mean of channels 1-5
```

This is followed by ';' for the end-of-sensor packet.

Status bit-flags are:

```
no3_ok       =      0, // no error
no3_parse    = 0x0001, // error parsing the return string
no3_multi    = 0x0002, // failed all tries to parse
no3_tmo      = 0x0004, // timed-out from inactivity
no3_busy     = 0x0008, // not correct state when asked for sample
no3_badStart= 0x0010, // bad start of string to parse
no3_err_ack = 0x0020, // no ack from 'W'
no3_bad_crc = 0x0040, // bad crc
no3_emu      = 0x0080, // emulation-mode
no3_tsdat    = 0x0100, // error on 'ACK,TS,DAT'
no3_err_sl   = 0x0200, // error on getting SL response
no3_err_ctd = 0x0400, // error on getting the ctd
no3_err_ts   = 0x0800, // error on getting 'ACK,TS,CMD'
no3_err_rtc = 0x1000, // error on settin the RTC
```

bistIDno3b = 8 spectrum

This sensor packet has the spectral values, S[k].
Dfat(0,4) **numSk** The number of spectral values.
The next **numSk** 2-byte (4-char) values are integer counts of each spectral bin.

bistIDno3c = 9 ascii string

This is an ascii string of the subset of the engineering values that the SUNA displays before the spectrum. These should match the values in bisIDno3a.

The string length is taken from the sensor packet length in the header, **<nDatSize>**, minus the header size and minus the trailing ';'.
nBytes = **<nDatSize>** - 7; // #of bytes in the sensor data payload.
nDatCh = nBytes*2; // # of hexadecimal characters for nBytes.
Dat(0, 2) = first byte value = ascii character
Dat(2, 2) = 2nd ascii character,
Etc. for nBytes.
The final ascii string will look like (begins and ends with ',')

```
,2240,0.02,21.2340,31.2340,470,551,20,23.12,23.19,3.06,12.83,0.574,0.00,0.00,
709,8,31.23,-67.21,1.283e-02,35,75,
```

With the arguments equal to:

```
1: 2240,       CTD timeStamp (epoch)
2: 0.02,       pressure
3: 21.2340,    temperature
4: 31.2340,    salinity
5: 470,        sample counter
6: 551,        power cycle counter
7: 20,         error counter
8: 23.12,      internal Temperature
9: 23.19,      Spectrometer T
10: 3.06,      Relative humidity
11: 12.83,     Supply Voltage
12: 0.574,     Supply current
13: 0.00,      Ref. Detector mean
14: 0.00,      Ref. Detector stdDev
15: 709,       Dark Spectrum Mean
16: 8,         Dark Spectrum stdDev
17: 31.23,     Sensor Salinity
18: -67.21,    Sensor Nitrate [uM]
19: 1.283e-02, Residual RMS
20: 35,        FIT Pixel Begin
21: 75,        FIT Pixel End
```

CONSTANT Definitions

This is a summary of defined constant values needed by the BGC data to rescale the data. They are represented using c code #define:

```
#define NSUB      16 // #points to pack per profile chunk

//=== parameter schedule array sizes  ====
#define N_SENSORS        6     // BGC max #sensors
#define PSM_MAX_VAR      2     // gain, offset for each sensor
#define PSM_MAX_SEN_VAR  5     // max individual parameters for each sensor
#define PSM_MAX_REG_VAR  5     // max # of variables per depth region
#define PSM_MAX_REGIONS  5     // max # of depth regions

//=== scalars used for auto-scaling ===
#define OCR_OFF_SCALAR 10000 // each OCR_off LSB = 10000 counts
#define  PH_OFF_SCALAR 20000 // each  PH_off LSB = 20000 uV = 0.02V

#define ECO_NVAR   3   // # of eco channels
#define DOX_NVAR   4   // # of variables in ea profile
#define OCR_NVAR   4   // # of variables in ea profile
#define  PH_NVAR   4   // # of variables in ea profile
#define NO3_NVAR  15   // max # variables for no3
#define NUM_SPEC  50   // max # of no3 spectra

//=== for parsing Engineering packets, hardwired gains: ===
#define CNTS_TO_DBAR  0.04  // convert p-counts to dBar
#define DBAR_OFFSET   10.0  // dBar = cnts*CNTS_TO_DBAR - DBAR_OFFSET
#define CNTS_TO_V14   0.01  // volts = cnts*CNTS_TO_V14
#define CNTS_TO_V07   0.01  // volts = cnts*CNTS_TO_V07
#define CNTS_TO_INHG  0.01  // in-Hg = cnts*CNTS_TO_INHG
#define CNTS_TO_MA  1 //  // mA = cnts*CNTS_TO_MA

/=== used in autogain for OCR and pH ===
#define OCR_OFF_SCALAR 10000 // each OCR_off LSB = 10000 counts
#define  PH_OFF_SCALAR 20000 // each  PH_off LSB = 20000 uV = 0.02V
#define Y2MAX   65534  // max counts range for unpacked unsigned 2-byte output

//=== pH and OCR offset and gain ===
#define  PH_V_OFF  2.5  // pH Voltage offset: shift -2.5 up to 2.5
#define  PH_V_GN   1e6  // turn V to uV counts
#define  OCR_COUNTS_DIV 1024  // first divide OCR counts by 1024
#define  OCR_COUNTS_OFF 0x1e0000 // and then subtract the offset  = 1966080
```

Four Bytes to floating-point: c-code example

```
float       int2float  (  unsigned int n  )  //
*******************************
{ // interpret the number n as a float
  // we have read in 4 bytes as an integer
  // but it is really a float.
  // this gets the byte-order correct for a float
  // and returns the float value
  int              j, k;
  unsigned char    t, b[4];
  float            *x;

  x = (float*) &b[0]; // address of x =start of byte array
  // fortran can use an equivalence statement instead


      for (j=0; j<4; j++ ) // -----------------
      { // convert the 4-byte n to 4-byte array
        if ( !LITTLE_ENDIAN) { k = j; } // b[0] = LSB
        else  { k = (3 - j); } // else b[3] = LSB
        t   = n & 0xff; // lsbyte
        b[k]  = t;  // b[0] = LSB for little-endian
        n = n >> 8; // right-shift to next byte
      } // ------------------------------------

      //for (j=0; j<4; j++ ) printf("%d:%02x\n", j, b[j] );
      //printf("LE = %d, x = %6.3f \n", LITTLE_ENDIAN, *x );

  return  *x;
} // ***************************************************************************
```

Float Code Examples

Setting the Sensor Packet Header

A struct is defined to allow easier setting of each of the header parameters:
(see c:\svn\spray2\code\shared\lifo.h)

```
//=== struct to handle sensor packet header ========================
#define PKT_HDR_SIZE  6 // #bytes in header
typedef struct { // header info
  int       sensorID;  // sensor family ID: will be stored in one byte b[0]
  int       nPktSize;  // #bytes in the packet: will be stored in 2bytes b[1]-b[2]
  int       packType;  // packing algorithm ID: stored in MSN b[3a]
  int       data_ID;  // data product ID: stored in LSN b[3b]
  int       proType;  // profile type: avg. scheme stored in MSN b[4a]
  int        proDir;  //  profile direction, stored in LSN b[4b]
  int         pSame;  // if >0, press series is same as another sensor. b[5a]
  int        subSeg;  // segment ID of subset of profile b[5b]

  int  elSize;  // for none-packed data, #bytes per datum (element)
  int   byte2; // = msn(proType) + lsn(proDir)
  int   byte3; // = msn(pSame)   + lsn(subSeg)
  //
 } pktHdr_t;
// ================================================================
```

The struct elements just need to be set for that sensor packet. Here is example code from C:\svn\spray2\code\shared\lifo.c

setSenHdr()

```
void        setSenHdr  ( sensorID_t  senID, // sensorID
                         int32_t      nDat,  // len of data packet, includes ';'
                         uint8_t      packType, // packing algorithm, type pack_t
                         uint8_t      datID, // data-product ID
                         uint8_t      proType, // profile-type, type proType_t
                         uint8_t      proDir,  // dir of profile, type proDir_t
                         uint8_t      pSame,  // associated p. channel
                         uint8_t      subSeg   ) // sub-segment ID
{ // store values to SenHdr
  clearSensorHeader(); // clear out any old hdr values
  // and set new values
  SenHdr.sensorID  = senID; // sensor ID
  SenHdr.nPktSize  = nDat + PKT_HDR_SIZE; // total sensor packet size
  SenHdr.packType  = packType; // packing algorithm
  SenHdr.data_ID   = datID; // data-product ID;
  SenHdr.proType   = proType; // profile type
  SenHdr.proDir    = proDir;  // direction of the profile
  SenHdr.pSame     = pSame;   // if applicable, points to another p array
  SenHdr.subSeg    = subSeg;
  packSensorHeader(); // pack the header struct into PktHdr bytes
  // now set to send!
```

```
    return;
  }  //  ***************************************************************************
```

Once the header struct is defined, it is packed into the packet header, PktHdr, via:

packSensorHeader()

```
int32_t      packSensorHeader  (  void  )  //  *******************************
{ // take the global sensor header SenHdr and pack it  into the
  // global PktHdr array
  // returns:
  //   k = # of bytes in the sensor header
  int32_t  k=0;

  PktHdr[k++] = SenHdr.sensorID & 0xff; // sensor ID
  k = b2_store( k, PktHdr, SenHdr.nPktSize ); // store #bytes
  PktHdr[k++] = pack_MSN_LSN( SenHdr.packType , SenHdr.data_ID  );
  PktHdr[k++] = pack_MSN_LSN( SenHdr.proType  , SenHdr.proDir   );
  PktHdr[k++] = pack_MSN_LSN( SenHdr.pSame     , SenHdr.subSeg   );
  // PktHdr is now complete.
  // it can be transferred to the LIFO followed by the  data + ';'

  return  k;
} //  ***************************************************************************
```

When a sensor packet is appended to a LIFO, the new function uses the local static of PktHdr:

lifo_Append()

```
void         lifo_Append  (  uint8_t *dat )  //  ******************************
{ // mar21 re-write: assumes senHdr, PktHdr are correct
  // and appends both the PktHdr and *dat to the lifo
  // Will start a new message if this over-runs the lifo
  int32_t    i,j, nb, bid, iprof, nPkt, nDat;
  uint8_t    *out;  // points to output location

  i    = lifo->k; // points to the present output lifo buffer
  nb   = lifo->nbytes[i]; // = #bytes presently in the block
  nPkt = SenHdr.nPktSize; // #of bytes in the total sensor packet
  nDat = nPkt - PKT_HDR_SIZE; // #of bytes in dat[]
  // deleting some lifo-overun code here so easier to follow
  out = lifo->buf + (long)i * (long)LIFO_NBYTE +nb; //=output location
  //--- xfr the header
  for ( j=0; j<PKT_HDR_SIZE; j++ )
  { *out++ = PktHdr[j]; }
  //--- xfr the data
  for ( j=0; j<nDat; j++ )
  { *out++ = *dat++;  }
  lifo->nbytes[i] = nb + nPkt; // adjust #bytes NOW in buffer
  // because we've added to it, make sure that its status is set back to "unsent"
  lifo->stat[i] = VIRGIN;

  return;
} // ****************************************************************************
```

Example

Here is an example for the Spray surface drift:

```
void         lifo_SurfDrift  (  bool  valid  )  //  ***************************
{ //  store surface drift information
  //  valid = #'s should be good
  uint8_t  dat[20];
  int32_t  i, n;

  i = 0;
  dat[i++] = (uint8_t) valid;  // = if valid or not
  i = b2_store(i,dat, Ublox.dt );
  i = b2_store(i,dat, Ublox.dx );
  i = b2_store(i,dat, Ublox.dy );
  dat[i++] = ';'; // end-of-packet char
  n = i; //=# of bytes of data
  // assign sensor header values
  setSenHdr( sen_NAV, n, pack_none, (uint32_t)navID_surfDrift,  0, 0, 0, 0);

  lifo_Append( dat ); // append to present lifo

  return;
} //*************************************************************************
```