Gemini Software Development Kit

Specification for Gemini Interface Library

(‘GeminiComms’)

Notes:

Gemini Sonar MK1/MK2: Applies to **VDSL**/**Ethernet** connections.
Gemini Sonar Micro platforms (720im and MicronGemini): Applies to **Serial/Ethernet** connections. These sonars are not fitted with velocimeter and all Gain functions not applicable in this version.
Gemini 1200ikd is a MK2 model which has dual frequency **720/1200** switching capability without VDSL support.
The ‘GeminiComms.dll’ (or Linux Equivalent) and relevant header files will be provided with this documentation.

Tritech

Revision History

Rev.	Date	Author	Changes
01	18/08/2009	SPG	Release of full interface specification.
02	10/09/2009	KM	Edited for SDK Development release.
03	02/03/2012 – 30/03/2012	PM	New GEMX interface to support multiple sonars. Added support for 720i(deep) and NBI products. Minor updates and corrections throughout.
07	08/04/2016	AS	Added support for 720iks (auxiliary)
08	10/10/2016	AS	Added support for Chirp and up to 512 beams for MK2 platform
09	17/05/2017	AS	Added support for 720iks and a table representing MK2 products
10	12/02/2018	AS	Added support for 720im
11	23/04/2019	KM/AS	Added support for 1200ikd
12	07/10/2019	AS	Replaced GEMX_ConfigureTransducerFrequency API with GEMX_ConfigureAutoRangeFrequency API to allow automatically switch between low/high frequency based on the range. Replaced GEMX_SetFilterBank with GEMX_ConfigureChirpMode Removed legacy API's
13	17/03/2021	AS	Updated description for GEMX_ConfigureMainPort API
14	18/02/2022	AS	Created a table for Gemini Micro platforms e.g. 720im and MicronGemini 720s
15	10/03/2022	AS	Added a new API to configure Noise reduction filter for Micron Gemini platform
16	15/02/2023	KM	Added clarification on GEMX_ConfigureMainPort regarding closing a port.
17	18/08/2023	MS	Added H264 Compression SDK functionality
18	09/05/2024	KM	New flag added in Status packet to indicate Ping Broadcasting switched on (720iks only). GEMX_SetAltSonarIPAddressWithFlag added to provide toggling of packet broadcasts.
19	04/04/2025	KM	Added support for 1200id hardReset argument added to GEM_StartGeminiNetworkWithResult

Index

Overview

This document describes the library which acts as the interface between the Gemini sonar head(s) and other systems, such as the Seanet Pro and Gemini software packages. It will provide details of all the library functions allowing users to gain programmatic access to the Gemini sonar in their own software. This document will describe all the functionality exposed by the library that is relevant for users to write their own control and display programs.

Only one application (and only one instance of that application) may use the Library at any given time to avoid resource conflicts.

The library is written in C++, using Microsoft Visual Studio 2017.

This document also describes Gemini SDK version V2.0.40 of the Gemini Interface Library, which returns the version string "`Gemini Comms V2.0.40 Tritech International Ltd.`"

Note on IP Addresses

To ensure that broadcast status messages from Gemini sonars are received, the library receives on the Gemini Rx port (52900) from any network adapter.

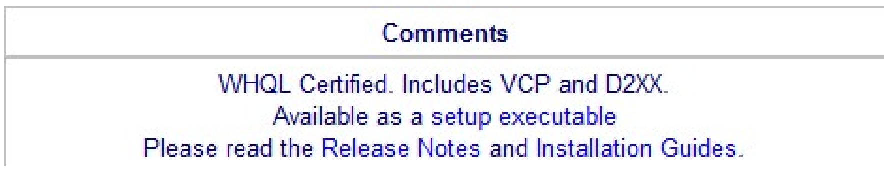
When transmitting to a Gemini sonar, the library transmits on the Gemini Tx port (52901) on the network adapter with an IP address that best matches that of the Gemini sonar.

It is assumed that in a multiple sonar system, all sonars will be set up on the same subnet, so that the library is only communicating with one network adapter.

Note for 720im and MicronGemini Sonars (See Table 5.0)

Switching between Ethernet to Serial interface requires power cycle to sonar

For operating Gemini Sonar devices with serial communications, please ensure FTDI driver has been installed from the following website: http://www.ftdichip.com/Drivers/D2XX.htm. Choose appropriate driver from the table for the table for the operating system of the PC. If available, choose the setup executable option:



Comments
WHQL Certified. Includes VCP and D2XX. Available as a setup executable Please read the Release Notes and Installation Guides.

Constants

The following messages are passed to the user defined data handling callback function along with the type of the data. Data types are defined later in this documents

PING_HEAD	0
PING_DATA	1
PING_TAIL	2
GEM_STATUS	3
GEM_ACKNOWLEDGE	4
** GEM_SERIAL	5
* GEM_FLASH_RESULT	6
GEM_BEARING_DATA	7
* GEM_FLASH_READBACK	8
* GEM_TEST_RESULT	9
PING_TAIL_EX	10
GEM_IP_CHANGED	11
GEM_UNKNOWN_DATA	12
* GEM_VELOCIMETER_DATA	13
* GEM_SERIAL_PORT_INPUT	14
* GEM_PROD_DATA	15
* GEM_GPI_DATA	16
* GEM_DIAG_DATA	17
* GEM_REMOTEPINGCOMMAND (obsolete)	18
* GEM_REMOTEPINGRESPONSE (obsolete)	19
* GEM_GPI_TIMING	20
* GEM_UDP_ACK	21
* GEM_DMG_DATA	22
* GEM_TS_ACK	23
** AZ_PING_HEAD	24
** AZ_PING_DATA	25
** AZ_PING_TAIL	26
** AZ_PING_TAIL_EX	27

Table 1.0 Message Type

`* Not required by user / ** Details not provided in this document.`

Only for MK2 platforms (See Table 4.0): Each message coming from the sonar contains a device ID. Both BF and DA send their own status messages. These device IDs are quite useful for identifying the device type sending the messages.

BEAMFORMER_FPGA	2
DATA_ACQ_FPGA_0	4
DATA_ACQ_FPGA_1	8
DATA_ACQ_FPGA_2	16

Table 2.0 Device IDs

The following table shows the supported Gemini product IDs. These product IDs are used with the function GEMX_SetHeadType to set up the library for the Gemini product in use.

720i	GEM_HEADTYPE_720I	0x0
720i (deep)	GEM_HEADTYPE_720ID	0x1
NBI	GEM_HEADTYPE_NBI	0x2
720is	GEM_HEADTYPE_MK2_720IS	0x21
1200ikd	GEM_HEADTYPE_MK2_1200IK	0x1D
720iks	GEM_HEADTYPE_MK2_720IK	0x1E
720im	GEM_HEADTYPE_720IM	0x1F
MicronGemini 720s	GEM_HEADTYPE_MICRON_GEMINI	0x27
1200id	GEM_HEADTYPE_MK2_1200ID	0x29

Table 3.0 Gemini Product ID

GEM_HEADTYPE_MK2_720IS
GEM_HEADTYPE_MK2_720IK
GEM_HEADTYPE_MK2_1200IK
GEM_HEADTYPE_MK2_1200ID

Table 4.0 MK2 Platforms

GEM_HEADTYPE_720IM
GEM_HEADTYPE_MICRON_GEMINI

Table 5.0 MICRO Platforms

Structures

The following structures are used in the interface of the library. The C++ class definitions for these are contained in the file ‘GeminiStructuresPublic.h’. Pointers to these structures are returned to the parent code through the callback function (see below).

CGemHdr

The CGemHdr structure is a common structure which is returned at the start of all data sent from the library to the parent software.

The fields within the CGemHdr structure are defined below

Field	Bytes	Definition
m_type	1	Identifies the type of data contained in the structure (see descriptions of individual structures for values)
m_version	1	Version number
m_deviceID	2	The unique ID of the sonar
m_packetLatency	2	Not currently implemented
m_src_sub_device_id	1	Only for MK2 platforms. 1 for downlink messages. For uplink messages, one of the device IDs from the Table 2.0.
m_dst_sub_device_id	1	Only for MK2 platforms. 1 for uplink messages. For downlink messages one or more device IDs from the Table 2.0.

CGemStatusPacket

In Gemini Mk1, the CGemStatusPacket is broadcast once per second by the Gemini Sonar head. In Gemini Mk2 platforms, 2 new structures are introduced: CGemMk2BFStatusPacket and CGemMk2DAStatusPacket. The status message is broadcast once per second from each device with its own device ID (See Table 2.0 for device IDs). The device ID informs that the message is coming from the beam former (BF) <u>or</u> the data acquisition (DA) board. The following table shows the difference of the status message:

Field	Bytes		Description		
		IM / Micron	Mk1	Mk2 – BF	Mk2 - DA
m_head	8	A CGemHdr structure – see definition of CGemHdr			
m_firmwareVer	2	The version of firmware in the Gemini sonar head			
m_sonarId	2	The sonar ID of the Gemini sonar head			
m_sonarFixIp	4	Fixed IP address of the sonar (192.168.2.200)			not used
m_sonarAltIp	4	Alternate IP address of the sonar (user programmed)			not used
m_surfaceIp	4	IP address of surface system currently connected to sonar			not used
m_flags	2	Bit 0 Serial message time stamp enable 0 - Time stamp disabled. 1 - Time stamp enabled and added to serial message. Bit 2 Ping Broadcasting (direct broadcasts) 0 – Disabled (normal for all Gemini models except 720is) 1 – Enabled (720is only) <u>Note</u>: When Ping Broadcasting is Enabled, the CGemPingHead, CGemPingLine & CGemPingTail packets with be direct broadcast			not used

		to destination address xxx.xxx.xxx.255 on the Class C subnet, e.g. if Gemini IP address is 192.168.2.201 then destination address with be 192.168.2.255.			
		Bits 15..8 Product ID See Table 3.0			
m_vccInt	2	FPGA VCC internal			not used
m_vccAux	2	FPGA VCC auxiliary			not used
m_dcVolt	2	not used	Incoming DC voltage	not used	not used
m_dieTemp	2	FPGA Die Temperature			not used
m_dipSwitch	2	Bit 1 0 = MS5837 Pressure Sensor Not Fitted 1 = MS5837 Pressure Sensor Fitted Bit 2 0 = XSENS Motion Sensor Not Fitted 1 = XSENS Motion Sensor Fitted		Bits (3 downto 0) DIP switch state	
m_vga1aTemp	2	TE0715 PCB Temp (10-bit unsigned, Kelvin*2)	VGA 1a Temperature	FPGA PCB Temp (12-bit unsigned)	FPGA PCB Temp (12-bit signed)
m_vga1bTemp	2	PSU/TX/IMU PCB Temperature	VGA 1b Temperature	HSC PCB Temp (12-bit signed)	AFE#0 – VGA Top Temp (12-bit signed)
m_vga2aTemp	2	AFE PCB Temperature	VGA 2a Temperature	Tx & VoS PCB (12-bit signed)	AFE#0 – VGA Bottom Temp (12-bit signed)
m_vga2bTemp	2	Transducer Temperature	VGA 2b Temperature	not used	AFE#1 – VGA Top Temp (12-bit signed)
m_psu1Temp	2	not used	PSU1 Temperature Leaded temperature probe, not present on production units.	PSU & Aux PCB (12-bit signed)	AFE#1 – VGA Bottom Temp (12-bit signed)
m_psu2Temp	2	not used	PSU2 Temperature This is the temperature sensor on the PSU board.	not used	AFE#2 – VGA Top Temp (12-bit signed)
m_currentTimestampL	4	32 bit timestamp (micro time) Sampled at the time of reading.			unused
m_currentTimestampH	4	32 bit timestamp (micro time) Sampled at the time of reading.			unused
m_transducerFrequency	2	0 = 720 1 = 620	0 = 868kHz 1 = 620kHz		unused
m_subnetMask	4	Subnet mask of the sonar			unused
m_TXTemp1	2	not used	TX Temp Sensor 1	not used	AFE#2 – VGA Bottom Temp (12-bit signed)
m_TXTemp2	2	MS5837 External Water Temperature (˚C) **((float)(ReturnValue-2000))/(100.0)** See Appendix C	TX Temp Sensor 2	not used	AFE#3 – VGA Top Temp (12-bit signed)
m_TXTemp3	2	MS5837 External Pressure (bar) **((300000/2^16)×ReturnValue)/1**	TX Temp Sensor 3	not used	AFE#3 – VGA Bottom Temp (12-bit signed)

		0000 See Appendix C			
m_BOOTSTSRegister	4		BOOTSTS register read from ICAP module. This contains the error codes for the last two boot attempts with 0 being the most recent. See Table 6.0 for each Bit details.		
m_shutdownStatus	2	Bit 0 – Over Temperature shutdown	Bit 0 – Over Temperature shutdown Bit 1 – Out of water shutdown Bit 2 – Out of Water indicator		Bit 0 – Over Temperature shutdown
m_dieOverTemp	2	Die Over Temperature threshold			not used
m_vga1aShutdownTemp	2	TE0715 PCB Temperature threshold	VGA 1a Temperature shutdown Threshold	FPGA PCB Over Temp Threshold	FPGA PCB Over Temp Threshold
m_vga1bShutdownTemp	2	PSU/TX/IMU PCB Temperature threshold	VGA 1b Temperature shutdown Threshold	HSC PCB Over Temp Threshold	AFE#0 – VGA Top Over Temp Threshold
m_vga2aShutdownTemp	2	AFE PCB Temperature threshold	VGA 2a Temperature shutdown Threshold	Tx & VoS PCB Over Temp Threshold	AFE#0 – VGA Bottom Over Temp Threshold
m_vga2bShutdownTemp	2	Transducer Temperature threshold	VGA 2b Temperature shutdown Threshold	not used	AFE#1 – VGA Top Over Temp Threshold
m_psu1ShutdownTemp	2	not used	PSU1 Temperature shutdown Threshold	PSU & Aux PCB Over Temp Threshold	AFE#1 – VGA Bottom Over Temp Threshold
m_psu2ShutdownTemp	2	not used	PSU2 Temperature shutdown Threshold	not used	AFE#2 – VGA Top Over Temp Threshold
m_TX1ShutdownTemp	2	not used	TX Temp 1 shutdown Threshold	not used	AFE#2 – VGA Bottom Over Temp Threshold
m_TX2ShutdownTemp	2	not used	TX Temp 2 shutdown Threshold	not used	AFE#3 – VGA Top Over Temp Threshold
m_TX3ShutdownTemp	2	not used	TX temp 3 shutdown Threshold	not used	AFE#3 – VGA Bottom Over Temp Threshold
m_linkType	2	Bit 0..7 unused Bit 9..8 – link speed (Ethernet) 00 – 10 Mbps 01 – 100 Mbps 10 – 1000 Mbps Bit 11..10 00 - No Link 01 – Ethernet 10 – RS232 11 – RS485	Bit 0 – link type 0 – Ethernet 1 – VDSL Bit 9..8 – link speed (Ethernet) 00 – 10 Mbps 01 – 100 Mbps 10 – 1000 Mbps		
m_VDSLDownstreamSpeed1	2	not used	Bit 7..0 – downstream 1 constellation Bit 15..8 – downstream 1 symbol rate		
m_VDSLDownstreamSpeed2	2	not used	Bit 7..0 – downstream 2 constellation Bit 15..8 – downstream 2 symbol rate		
m_macAddress1	2	Least significant two bytes of Sonar head MAC address			
m_macAddress2	2	Middle two bytes of Sonar head MAC address			
m_macAddress3	2	Most significant two bytes of Sonar head MAC address			
m_VDSLUpstreamSpeed1	2	Bit 31..0 Main port serial rate	Bit 7..0 – upstream 1 constellation Bit 15..8 – upstream 1 symbol rate		
m_VDSLUpstreamSpeed2	2		Bit 7..0 – upstream 2 constellation Bit 15..8 – upstream 2 symbol rate		

The following table defines the m_BOOTSTSRegister field.

BOOTSTS register read from ICAP module. This contains the error codes for the last two boot attempts with 0 being the most recent.

m_BOOTSTSRegister			
Bits	**MK1**	**MK2-BF**	**MK2-DA**
0	0_Valid	0_Valid	0_Valid
1	0_Fallback	0_Fallback	0_Fallback
2	0_IPROG	0_IPROG	Reserved
3	0_WTO_ERROR	0_WTO_ERROR	0_WTO_ERROR
4	0_ID_ERROR	0_ID_ERROR	0_ID_ERROR
5	0_CRC Error	0_CRC Error	0_CRC Error
6	0_BPI Address wraparound error	0_WRAP Error	1_Valid
7	0_RBCRC error caused reconfiguration	Reserved	1_Fallback
8	1_Valid	1_Valid	Reserved
9	1_Fallback	1_Fallback	1_WTO_ERROR
10	1_IPROG	1_IPROG	1_ID_ERROR
11	1_WTO_ERROR	1_WTO_ERROR	1_CRC Error
12	1_ID_ERROR	1_ID_ERROR	STRIKE CNT 0
13	1_CRC Error	1_CRC Error	STRIKE CNT 1
14	1_BPI Address wraparound error	1_WRAP Error	STRIKE CNT 2
15	1_RBCRC error caused reconfiguration	Reserved	STRIKE CNT 3

Table 6.0 Boot Status Register

For the CGemStatusPacket structure, the value of `m_head.m_type` is 0x40.

Due to the layout of the Gemini Status Packet, the C code for the structure aligns it on a two byte boundary rather than the normal four byte boundary. This will also need to be implemented in any code which maps onto the structure returned by the library.

The following conversion gives the FPGA die temperature value in °C…

```
FPGA_Die_Temp = ((m_dieTemp * 503.975) / 1024.0) - 273.15
```

The upstream and downstream data rates (measured in bits per second) are given by…

```
Current data rate downstream =
(Downstream 1 constellation *
 Downstream 1 symbol rate * 67500) +
(Downstream 2 constellation *
 Downstream 2 symbol rate * 67500)

Current data rate upstream =
(Upstream 1 constellation *
 Upstream 1 symbol rate * 67500) +
(Upstream 2 constellation *
 Upstream 2 symbol rate * 67500)
```

For the PSU temperature, the four VGA temperatures, the three transmitter temperatures and the matching shutdown temperatures, the value returned is made up as follows…

bit 15	If set, the temperature sensor is present
bit 10	If set, the temperature is negative
bits 9 to 0	The temperature value

The following code converts a temperature value to a string, taking the above into account (the function `StringCbPrintf` is like the normal C `printf` function, apart from it produces wide format output strings)…

```
void TempStr(unsigned short temperature, TCHAR *outStr)
{
  signed short tempTemp;
  double       finalTemp;

  // If sixteenth bit is set, sensor is present
  bool present = ((temperature & 0x8000) == 0x8000);

  if (present)
  {
    // If tenth bit is set, value is negative
    bool negFlag = ((temperature & 0x0200) == 0x0200);

    // Temperature is contained in 10 bits of value
    tempTemp = temperature & 0x3ff;

    // Sign extend value so that it is a C style signed number
    if (negFlag)
    {
      tempTemp |= 0xfc00;
    }

    // Convert to floating point
    finalTemp = (double)tempTemp / 4.0;

    // And convert to a wide string
    StringCbPrintf(outStr, 20, _T("%1.2lf"), finalTemp);
  }
  else
  {
    // Indicate sensor not present
    StringCbPrintf(outStr, 20, _T(" N/P "));
  }
}

static double TempStr MK2Platforms(unsigned short temperature )
{
  signed short tempTemp;
  double       finalTemp = 0;

  // If twelfth bit is set, value is negative
  bool negFlag = ((temperature & 0x0800) == 0x0800);

  // Temperature is contained in 12 bits of value
  tempTemp = temperature & 0xfff;

  // Sign extend value so that it is a C style signed number
  if (negFlag)
  {
    tempTemp |= 0xf000;
  }

  // Convert to floating point
  finalTemp = (double)tempTemp * 0.0625;

  return finalTemp;
}

void DisplayTemperatureFunction(unsigned short vga1aTemp)
{
  TCHAR tStr[20];

  TempStr(vga1aTemp, tStr);

  // Do something to display the string returned by TempStr here
}
```

CGemPingHead

The fields within the CGemPingHead structures are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_pingID	2	A 16 bit ping id which ties together the ping header, data and tail. Wraps round. **Bit 7..0** **–** 8 bit Ping ID **Bit 8** **–** Using Transducer (0 : lower frequency, 1 : higher frequency) **Bit 13** **–** Using flash modifiers toggle **Bit 14** **–** Swap Double Buffers in acted toggle **Bit 15** **–** Denotes re-sent ping header
m_extMode	2	The value of the extMode field sent in the ping configuration.
m_transmitTimestampL	4	Transmit time in microseconds – lower 32 bits
m_transmitTimestampH	4	Transmit time in microseconds – upper 32 bits
m_startRange	2	Start range (in lines) of the data being returned
m_endRange	2	End range (in lines) of the data being returned
m_lineTime	4	Not currently implemented
m_numBeams	2	Number of beams formed in the data being returned
m_numChans	2	Number of channels in the data being returned
m_sampChan	1	Value of sample channel sent in the ping configuration
m_baseGain	1	Value of base gain sent in the ping configuration. **Note**: Not implemented in this version for Gemini Micro platforms (See Table 5.0). **Bits 1..0 –** base gain **Bits 7..6 –** Data Acq. Number (0=DA#0, 1=DA#1, 2=DA#2)
m_spdSndVel	2	Speed of sound being measured by the velocimeter in units of 0.1m/s. For Gemini models not fitted with a velocimeter, the fixed value being used will be returned.
m_velEchoTime	2	Time for velocimeter echo – used for calibration only
m_velEntries	2	Number of velocimeter readings used in speed of sound calculation
m_txAngle	2	Not currently implemented
m_sosUsed	2	Speed of sound used in the beamforming. Bit 15 is used to indicate the source of the speed of sound Bit 15 = 0, the speed of sound came from the velocimeter Bit 15 = 1, the speed of sound came from the ping config Bits 14 to 0 carry the speed of sound in units of 0.1m/s
m_RLEThresholdUsed	1	Threshold value applied to Run Length Encoding – zero implies no RLE applied. The library will remove any RLE applied before the data is passed to the parent application.
m_rangeCompressionUsed	1	Value indicating what range compression, if any, was applied to the data. The library may remove any range compression applied before the data is passed to the parent application. This is dependent on the mode the library is set to. **Bit 3..0 – Range compression level** 0000 - No compression 0001 - 2 x Compression 0010 - 4 x Compression 0011 - 8 x Compression 0100 - 16 x Compression **Bit 4 – Range Compression type** 0 – Average compression 1 – Peak compression
m_internal	1	Internal value, ignore.

For the CGemPingHead structure, the value of `m_head.m_type` is 0x41.

CGemPingLine

The fields within the CGemPingLine structure are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_gain	1	Not currently implemented
m_pingID	1	Lower 8 bits of the ping id which ties together the ping header, data and tail. Wraps round.
m_lineID	2	The id of the line within the ping data (value ranges from 0 to (CGemPingHead::m_endRange – CGemPingHead::m_startRange) – 1. Bit 15 is used to indicate that this is resent data (part of the retry mechanism implemented by the library)
m_scale	2	Value of software gain applied to this line
m_lineInfo	2	Bits 5..0 Ext_mode from ping config bits 5..0 Bit 6 Last Line Flag (1 indicates last line) Bits 15..7 Number of bytes in data
* m_startOfData	1	Field which gives start address of data within the structure

For the CGemPingLine structure, the value of `m_head.m_type` is 0x42.

* `m_startOfData` is the first byte of the data being returned to the parent process and therefore is the first byte of an array of `CGemPingHead::m_numBeams` of data.

The library implements a retry mechanism to re-request data lost due to dropped packets. Bit 15 (MSB) of `m_lineID` is used to indicate if a packet is carrying data as a result of a retry request. The library will strip this bit out before the data is passed to the calling program and so will always be seen as zero at the calling program.

CGemPingTail

The fields within the CGemPingTail structure are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_pingID	1	Lower 8 bits of the ping id which ties together the ping header, data and tail. Wraps round.
m_flags	1	Bit 7 is used to indicate that this is resent data (part of the retry mechanism implemented by the library).
m_spare	2	Not used

For the CGemPingTail structure, the value of `m_head.m_type` is 0x43.

The library implements a retry mechanism to re-request data lost due to dropped packets. Bit 7 (MSB) of `m_flags` is used to indicate if a packet is carrying data as a result of a retry request. The library will strip this bit out before the data is passed to the calling program, and so will always be seen as zero at the calling program.

CGemPingTailExtended

The CGemPingTailExtended structure is generated within the library when a ping tail message is received from the Gemini Sonar. As well as the fields received from the Sonar (which would be returned in a CGemPingTail structure), this structure also contains extra data generated by the library during the receipt of the ping data. This data typically contains the values of counters which help to evaluate the quality of the link to the Sonar and is described in more detail in the table below.

The fields within the CGemPingTailExtended structure are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_pingID	1	Lower 8 bits of the ping id which ties together the ping header, data and tail. Wraps round.
m_flags	1	Bit 7 is used to indicate that this is resent data (part of the retry mechanism implemented by the library).
m_spare	2	Not used
m_firstPassRetries	2	The number of first retries of data lines attempted
m_secondPassRetries	2	The number of second (or more) retries of data lines attempted
m_tailRetries	2	The number of retries of the ping tail attempted
m_interMessageGap	2	The value of the intermessage gap being set by the library when setting up communications with the sonar.
m_packetCount	4	The number of packets received since the library started receiving data. Excludes status packets.
m_recvErrorCount	4	The number of times the “recv” function in the library has returned an error.
m_linesLostThisPing	4	The number of lines of data which were expected but have not been received in the ping which has just occurred.
m_generalCount	4	A general count used to convey information from the library to the calling program. No specific use in this version of the library.

For the CGemPingTailExtended structure, the value of `m_head.m_type` is 0x43.

The library implements a retry mechanism to re-request data lost due to dropped packets. Bit 7 (MSB) of `m_flags` is used to indicate if a packet is carrying data as a result of a retry request. The library will strip this bit out before the data is passed to the calling program, and so will always be seen as zero at the calling program.

CGemAcknowledge

The fields within the CGemAcknowledge structure are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_receiptTimestampL	4	The time of receipt of the stay alive packet by the Sonar – Lower 32 bits
m_receiptTimestampH	4	The time of receipt of the stay alive packet by the Sonar – Upper 32 bits
m_replyTimestampL	4	Time of transmission – Lower 32 bits
m_replyTimestampH	4	Time of transmission – Upper 32 bits

For the CGemAcknowledge structure, the value of `m_head.m_type` is 0x49.

This message is sent in response to a keep alive packet.

CGemBearingData

The fields within the CGemBearingData structure are defined below.

Field	Bytes	Definition
m_head	8	A CGemHdr structure – see definition of CGemHdr
m_bearingLineNo	2	The bearing line number for this block of data (between 0 and number of beams)
m_noSamples	2	The number of samples in this block of data.
m_pData	variable	Address of the data – Block of separately allocated memory of size m_noSamples, which will be freed by the library.

For the CGemBearingData structure, the value of `m_head.m_type` is 0x60.

`m_data1` is a pointer to the first byte of the data being returned to the parent process and therefore is a pointer to the first byte of an array of `m_noSamples` of data.

Data is returned to the calling program via Gemini Bearing Data packets when the software is put into `SeaNet` or `SeaNetC` mode.

See the note about memory allocation and the CGemBearingStructure in the next section in the description of the main callback function.

Callback Functions

The library requires a callback function to be defined for it, so that it can inform the calling process when data has been received from the sonar head.

Main Callback Function

The main callback function has to have the following definition

```
void CallBackFn(int eType, int len, char *dataBlock)
```

`eType` is the type of data being passed from the library to the calling program. It is one of the previously defined constants, e.g. `PING_HEAD` or `GEM_STATUS`.

`len` is the length of the data block being passed to the calling program. For any blocks defined as fixed length in the table below, the value of `len` returned by the library is -1. For the variable length blocks, length is defined as follows.

Structure type	**Value of len**
CGemPingHead	-1
CGemPingLine	Number of beams in the record
CGemPingTail	-1
CGemPingTailExtended	-1
CGemStatusPacket	-1
CGemAcknowledge	-1
CGemBearingData	Number of lines in the record

`dataBlock` is the actual data being passed to the calling program, and is a pointer to one of the structures defined previously.

eType	**dataBlock is a**	**Fixed Length?**
PING_HEAD	CGemPingHead	✓
PING_DATA	CGemPingLine	×
PING_TAIL	CGemPingTail	✓
PING_TAIL_EX	CGemPingTailExtended	✓
GEM_STATUS	CGemStatusPacket	✓
GEM_ACKNOWLEDGE	CGemAcknowledge	✓
GEM_BEARING_DATA	CGemBearingData	×
GEM_UNKNOWN_DATA	Unknown	×
GEM_IP_CHANGED	Empty	✓
** AZ_PING_HEAD	CGemAZPingHead	✓
** AZ_PING_DATA	CGemPingLine	×
** AZ_PING_TAIL	CGemPingTail	✓
** AZ_PING_TAIL_EX	CGemPingTailExtended	✓
** AZ_GEM_BEARING_DATA	CGemBearingData	×

** Details not provided in this document.

The callback function is set up by calling the library function GEM_SetHandlerFunction as follows…

```
GEM_SetHandlerFunction(CallBackFn);
```

The design of the library requires that the callback function copies the data from `dataBlock` into local storage under its own control before returning, as the buffer holding the data in the library may be overwritten by other data coming from the sonar head if the block is used by the parent process after the callback function has returned.

For all structures where the length of the structure is not known at design time, the `len` field contains the length information, allowing the parent code to allocate the correct amount of memory to copy all the data.

Where multiple Gemini sonars are in use, the callback function must examine the `CGemHdr::m_deviceID` field in incoming messages to determine which Gemini sonar originated the message. Note that Gemini sonar Status messages are always passed to the callback function, regardless of whether the sonar that originated them is currently active.

For a GEM_BEARING_DATA message, the actual data contained in the message is passed in a block of memory allocated by the library. For each bearing line in a ping, the same block of memory is used for the data, and this memory is freed by the library after the last line of data has been sent. The calling program will need to copy the data out of this block of memory into memory allocated by the calling process (instead of just copying and retaining the value of the pointer).

For a GEM_IP_CHANGED message, there is no associated data. The library has a task which is notified if the IP table of the PC changes. The expectation is if this message is received, the calling program will shutdown and restart the communications with the Gemini sonar, thus allowing the change in the IP address of the computer to be correctly actioned.

The following snippet of code shows the minimum code expected to be executed when this message is received.

```
GEM_StopGeminiNetwork();
GEM_StartGeminiNetworkWithResult(0);
GEM_SetGeminiSoftwareMode("RequiredMode");
GEM_SetHandlerFunction(&GemDatahandler);
```

For a GEM_UNKNOWN_DATA message, there is a message block attached whose total length is returned in the `len` field. The Gemini library is not aware of what the packet is, so has passed the packet untouched to calling program.

Functions

The following functions are defined as the interface of the library. The C definitions for these are contained in the file ‘GeminiCommsPublic.h’. The library itself is known as GeminiComms.dll for Windows platform and libGeminiComms.so for Linux platform

For each of the functions exposed by the library, the following paragraphs give the name of the function, the C definition of the function, a description of the function, its parameters and return value (if any), an example of calling the function in C, and a summary of the default values and limits for the parameters (where applicable).

Functions are split into two groups:

Global functions

These functions have global control of the library and do not relate to a specific sonar ID.

Sonar specific functions

These functions relate to a specific `sonarID`. The library places no limit on the maximum number of sonars that can be supported; this is determined by availability of resources (memory, network bandwidth etc.) on the PC in use. The sonars may be any combination of 720i, 720i(deep), 720is, 720iks, 720im, 1200ikd, MicronGemini 720s & 1200id.

Previous versions of the SDK

Programs written with early versions of the Gemini SDK did not have access to the GEMX_ function interface to the library, and so only one Gemini sonar could be used at a time. The GEMX_ functions replace earlier functions as shown in the example below to allow more than one Gemini sonar to be used at a time:

Old single sonar function `void GEM_Function(int param);`
New multiple sonar function `void GEMX_Function(unsigned short sonarID, int param);`

For compatibility with existing application programs, the old functions are still included in the SDK. However, use of them for new application programs is strongly discouraged. By defining the following macro before including the `GeminiCommsPublic.h` header file, the old function declarations can be excluded so that an application program attempting to use them will fail to compile:

```
#define GEM_EXCLUDE_SINGLESONAR
```

Global Functions

These functions have global control of the library and do not relate to a specific sonar ID.

GEM_StartGeminiNetworkWithResult

```
int GEM_StartGeminiNetworkWithResult(unsigned short sonarID, bool
hardReset = false);
```

This function initialises the library into a state ready to operate.

`sonarID` must always be set to 0 when calling this function.
`hardReset` forces a reset of the internal Hardware Abstraction Layer - should never be need to be changed from default = false. The only use case is for reloading/resetting the GeminiComms library.

The return value is defined in the following table…

Return Value	Meaning
0	A failure occurred initialising the library and communication with the sonar is not possible. The most likely cause is that another piece of software using the Gemini library is already running, and so the port could not be opened.
1	The library initialised correctly and is ready to communicate.

```
int success = 0;
success = GEM_StartGeminiNetwork(0);

if (success == 0)
{
    // Tell the user that something went wrong
    // Either shutdown the program or
    // enter an offline mode
}
else
{
    // Hooray! We’re good to go...
}
```

GEM_SetGeminiSoftwareMode

```
void GEM_SetGeminiSoftwareMode(char *softwareMode);
```

This function sets the operating mode of the software. Setting a value other than those listed in the table below will result in the software mode being set to the default mode.

Notes:

`"EvoC"` is the Mode used by the Gemini software Package.
`"SeanetC"` is the Mode used by the Seanet Pro software Package.

softwareMode	Meaning
"Evo" (Deprecated)	The Gemini library sends the data from the Gemini sonar head to the calling program unprocessed as CGemPingHead, CGemPingLine and CGemPingTailExtended messages using the callback function.
"EvoC"	The Gemini library sends the data from the Gemini sonar head to the calling program unprocessed as CGemPingHead, CGemPingLine and CGemPingTailExtended messages using the callback function. The Gemini library makes use of the range compression feature of the sonar head firmware to ensure that the head does not return more range lines than is appropriate for the size and quality of the display being used by the calling program.
"SeaNet" (Deprecated)	The Gemini library processes the data and sends it to the calling program as CGemPingHead and CGemBearingData messages using the callback function.
"SeaNetC" (Deprecated)	The Gemini library processes the data and sends it to the calling program as CGemPingHead and CGemBearingData messages using the callback function. The Gemini library makes use of the range compression feature of the sonar head firmware to ensure that the head does not return 1500 or more range lines to the Seanet software.

```
GEM_SetGeminiSoftwareMode("SeaNet");
```

Default value: `"EvoC"`

GEM_SetHandlerFunction

`void GEM_SetHandlerFunction(void (cdecl *FnPtr)(int eType, int len, char *dataBlock));`

This function allows the parent code to specify the callback function which will be used by the library to return data from the sonar head to the parent code.

`FnPtr` is the address of the callback function in the parent code which the library will use.

```
void CallBackFn(int eType, int len, char *dataBlock)
{
  CGemStatusPacket *pStat;
  CGemStatusPacket *copyStatus;
  CGemBearingData  *pBearing;
  CGemBearingData  *copyBearing;

  switch (eType)
  {
    case PING_HEAD :
    case PING_DATA :
    case PING_TAIL :
      ...
    case GEM_STATUS :
      // Copy the data, so that we do something with it

      pStat = (CGemStatusPacket *)dataBlock;

      copyStatus = new CGemStatusPacket;

      *copyStatus = *pStat;

      // Do something with the message, remember to delete copyStatus

      break;

    case GEM_ACKNOWLEDGE :
      ...
    case GEM_BEARING_DATA :
      // Copy the data, so that we can post a message

      pBearing = (CGemBearingData *)dataBlock;

      copyBearing = new CGemBearingData;

      *copyBearing = *pBearing;

      // The bearing data structure contains a pointer to a block of memory which is
      // allocated in the library and which is filled with the data we want to display.
      // Therefore we cannot just take a copy of the structure, as this contains the
      // pointer to the memory allocated by the library.

      // Attempt to allocate the memory for the copied data
      copyBearing->m_pData = (unsigned char *)malloc(copyBearing->m_noSamples);

      // Did we allocate the memory?
      if (copyBearing->m_pData)
      {
        // Copy the data from the original structure to the new structure
        memcpy(copyBearing->m_pData, pBearing->m_pData, copyBearing->m_noSamples);
      }
      else
      {
        // Failed to allocate memory? Indicate by setting number of samples to zero
        copyBearing->m_noSamples = 0;
      }

      // Do something with the message, remember to free copyBearing->m_pData and copyBearing

      break;
  }
}

GEM_SetHandlerFunction(CallBackFn);
```

Default value: `null`

GEM_ResetInternalCounters

```
void GEM_ResetInternalCounters(void);
```

This function resets some internal counters held by the GeminiComms library, which are returned in the CGemPingTailExtended structure (a PING_TAIL_EX message).

The library keeps a number of counters, such as the packet received and retry counts that only it is aware of. These counters are from the time the library started running. There are a number of reasons why these counters may need to be reset, such as changing the sonar you are talking to in a multi-sonar environment. This function allows the calling program to reset the counters when it needs to.

```
GEM_ResetInternalCounters();
```

GEM_StopGeminiNetwork

```
void GEM_StopGeminiNetwork(void);
```

This function stops the functionality of the GeminiComms library, which among other things will stop the tasks running which are handling the network data and free any allocated memory. This function will sleep for a short period to allow the tasks running in the library to finish in an orderly fashion.

```
GEM_StopGeminiNetwork();
```

GEM_GetDLLLinkVersionNumber

```
unsigned char GEM_GetDLLLinkVersionNumber(void);
```

This function returns the maximum version of the data link protocol (`CGemHdr::m_version`) that the library can support. Currently this is 2.

```
unsigned char linkVer = GEM_GetDLLLinkVersionNumber();
```

GEM_GetVStringLen

```
int GEM_GetVStringLen(void);
```

This function returns the length of the version string which identifies the library.

```
int i = GEM_GetVStringLen();
```

GEM_GetVString

```
void GEM_GetVString(char *data, int len);
```

This function returns the version string which identifies the library. The function will return a maximum of `len` characters in the buffer pointed to by `data`.

```
int i = GEM_GetVStringLen();
char *p = malloc(i + 1);
GEM_GetVString(p, i);

// p points to memory containing the string
// 'Gemini Comms V1.07.71 Tritech International Ltd.'
```

GEM_GetVNumStringLen

```
int GEM_GetVNumStringLen(void);
```

This function returns the length of the version number string which identifies the library.

```
int i = GEM_GetVNumStringLen();
```

GEM_GetVNumString

```
void GEM_GetVNumString(char *data, int len);
```

This function returns the version number string which identifies the library. The function will return a maximum of `len` characters in the buffer pointed to by `data`.

```
int i = GEM_GetVNumStringLen();
char *p = malloc(i + 1);
GEM_GetVNumString(p, i);

// p points to memory containing the string
// '1.07.71'
```

GEMX_GetSonars

```
unsigned short GEMX_GetSonars(unsigned short *pList);
```

This function returns a count of the total number of sonars that the library is managing.

`pList` is a pointer to memory (allocated by the calling function) where a table of unsigned short values will be written by GEMX_GetSonars. If this is not required, set `pList` to `NULL`.

If `pList` is not `NULL`, then GEMX_GetSonars will write a table of the sonar IDs that it is managing at this address. The calling function must allocate memory for this and pass a pointer to that memory to GEMX_GetSonars.

Example usage:

```
unsigned short nSonars = GEMX_GetSonars(NULL);
unsigned short* pList = (unsigned short*)malloc(nSonars *
sizeof(unsigned short));
if (pList != NULL)
{
  GEMX_GetSonars(pList);
  ... use the list of Sonar IDs ...
  free(pList);
}
```

GEMX_DeleteSonarID

```
unsigned short GEMX_DeleteSonarID(unsigned short sonarID);
```

This function instructs the library to delete all data that it holds for the given `sonarID`, and to stop managing that sonar. Except for status messages, data received from that sonar ID will no longer be passed to the callback function.

If `sonarID` is 0, then the library will delete data for all sonar IDs that it is managing.

Example usage:

```
GEMX_DeleteSonarID(sonarID);
```

Return value: Count of the number of sonars deleted by this call.

Sonar Specific Functions

These functions relate to a specific `sonarID`. Each Gemini sonar has a unique ID and will only accept commands which have a matching ID to its own. The unique ID is printed on the rear of the sonar, and is broadcast in status messages from that sonar.

GEMX_SetHeadType

```
void GEMX_SetHeadType(unsigned short sonarID, unsigned int headType)
```

Call this function at the start of initialisation to configure the library for the correct sonar head type. `headType` must be one of the GEM_HEADTYPE_XXXX values defined in the header file GeminiStructuresPublic.h.

Default value: `GEM_HEADTYPE_720I`

```
GEMX_SetHeadType(sonarID, GEM_HEADTYPE_NBI);
```

It is important that this function is called at the start of initialisation, as the behaviour of the library and many other functions is altered depending on the type of sonar head being used. See Table 3.0 “Gemini Product ID”.

GEMX_GetAltSonarIPAddress

```
void GEMX_GetAltSonarIPAddress(unsigned short sonarID, unsigned char
*a1, unsigned char *a2, unsigned char *a3, unsigned char *a4,
unsigned char *s1, unsigned char *s2, unsigned char *s3, unsigned
char *s4);
```

This function gets the value the library holds for the alternative IP address and the subnet mask that the Gemini sonar can have. A Gemini sonar head will always respond to the IP address 192.168.2.200, but to allow easy integration onto other networks an alternative IP address can be specified which the Gemini head will also respond to. A subnet mask can also be specified to limit the interaction between multiple units on the same physical network. This function returns the alternative IP address which the library may use to communicate with the head.

```
unsigned char a1, a2, a3, a4; // IP Address, A1 is MSB
unsigned char s1, s2, s3, s4; // Subnet mask, S1 is MSB

GEMX_GetAltSonarIPAddress(sonarID,
                          &a1, &a2, &a3, &a4,
                          &s1, &s2, &s3, &s4);
```

GEMX_SetAltSonarIPAddress

```
void GEMX_SetAltSonarIPAddress(unsigned short sonarID, unsigned char
a1, unsigned char a2, unsigned char a3, unsigned char a4 unsigned
char s1, unsigned char s2, unsigned char s3, unsigned char s4);
```

This function sets the alternative IP address and the subnet mask that the Gemini sonar can have. A Gemini sonar head will always respond to the IP address 192.168.2.200, but to allow easy integration onto other networks an alternative IP address can be specified which the Gemini head will also respond to. This function programs the alternative IP address into the head.

The alternate address used by the library is not changed as the Sonar head will not use the newly programmed address until it is rebooted. After issuing this command, the calling program should wait for a fixed (worst case) period of 5 seconds before issuing a command to reboot the sonar. Rebooting the sonar before this command has completed will result in corrupted flash in the Sonar head, and an unusable Sonar.

```
// Set address 192.168.1.10, subnet mask 255.255.255.0
GEMX_SetAltSonarIPAddress(sonarID,
                          192, 168, 1, 10,
                          255, 255, 255, 0);
```

Default value: 0, 0, 0, 0, 0, 0, 0, 0

GEMX_SetAltSonarIPAddressWithFlag

```
void GEMX_SetAltSonarIPAddressWithFlag(unsigned short sonarID,
unsigned int ipAddress, unsigned int subnetMask, unsigned int
broadcastStatus, unsigned int broadcastPing);
```

This is an alternative function to GEMX_SetAltSonarIPAddress() to be used when toggling the Packet broadcast options. Normally, Status packet broadcasting will be Enabled (param value = 0x0) but the ‘broadcastStatus’ param can be set to a value of 0x01234567 to <u>Disable</u> these broadcasts. Ping Broadcasting (uses direct broadcasting with dest. addr. = xxx.xxx.xxx.255) is only currently available with Gemini 720is (requires BF Firmware v20.26) and the ‘broadcastPing’ param can be set to a value of 0x01234567 to <u>Enable</u> broadcasting.

Like SetAltSonarIPAddress(), this function sets the alternative IP address and the subnet mask that the Gemini sonar can have. A Gemini sonar head will always respond to the IP address 192.168.2.200, but to allow easy integration onto other networks an alternative IP address can be specified which the Gemini head will also respond to. This function programs the alternative IP address into the head.

Alternative IP Address and Subnet Mask parameters are passed as 32-bit Unsigned Integers in this function derivative. The broadcastStatus and broadcastPing parameters are also 32-bit Unsigned Integer values; default broadcastStatus = 0x0 (Enabled), default broadcastPing = 0x0 (Disabled).

```
// Set address 192.168.1.10, subnet mask 255.255.255.0, Status
broadcasting enabled, Ping broadcasting enabled
GEMX_SetAltSonarIPAddressWithFlag(sonarID,
                          3232235786, 4294967040,
                          0, 19088743);
```

GEMX_UseAltSonarIPAddress

```
void GEMX_UseAltSonarIPAddress(unsigned short sonarID, unsigned char
a1, unsigned char a2, unsigned char a3, unsigned char a4 unsigned
char s1, unsigned char s2, unsigned char s3, unsigned char s4);
```

This function informs the library of the alternative IP address and subnet mask that the Gemini sonar can have. A Gemini sonar head will always respond to the IP address 192.168.2.200, but to allow easy integration onto other networks an alternative IP address can be specified which the Gemini head will also respond to. This function informs the library of the alternative address which can be used to talk to the head.

```
// Use address 192.168.1.10, subnet mask 255.255.255.0
GEMX_UseAltSonarIPAddress(sonarID,
                          192, 168, 1, 10,
                          255, 255, 255, 0);
```

GEMX_TxToAltIPAddress

```
void GEMX_TxToAltIPAddress(unsigned short sonarID, int
useAltIPAddress);
```

This function tells the library to use either the fixed (default) or the alternate IP address when talking to the Gemini sonar.

useAltIPAddress	Meaning
0	Use fixed IP address to communicate with head (192.168.2.200)
1	Use alternative IP address to communicate with head

```
GEMX_TxToAltIPAddress(sonarID, 0);
```

GEMX_AutoPingConfig

```
void GEMX_AutoPingConfig(unsigned short sonarID, float range,
unsigned short gain, float sos);
```

This function tells the library to build a ping configuration packet using the range, gain and speed of sound specified in the function call, then calculating all other values for the ping from the range, the gain, and its internal defaults. The ping will be built so that the sonar head pings once in response. The ping configuration packet is sent to the Gemini sonar head when GEMX_SendGeminiPingConfig is called.

`range` is the range (in metres) which the sonar is expected to return data in this ping.

`gain` is the percentage gain which the sonar is to apply in this ping.

`sos` is the speed of sound (in m/s) to be used when calculating values for this ping. As with all speed of sound values associated with the Gemini sonar, for Mk1 sonars this value will be rounded to the nearest 3.2m/s step when it is used.

```
GEMX_AutoPingConfig(sonarID, 11, 50, 1473.6);
```

Default value:	`range`	`0`
	`gain`	`12`
	`sos`	`1499.2`
Minimum:	range	0m
	gain	12%
	sos	1400.0
Maximum:	range	120m
	gain	100%
	sos	1588.8

The minimum and maximum speed of sound values given above apply to the Gemini imager only.

GEMX_SetGeminiEvoQuality

```
void GEMX_SetGeminiEvoQuality(unsigned short sonarID, unsigned char
evoQualitySetting);
```

This function sets the compression factor applied when the library is running in `EvoC` mode. The Gemini sonar can range compress the data, so that less bandwidth is used in sending the data from the sonar head to the surface computer. This function matches the range compression applied to the quality setting of the Evo system. The Evo system quality setting defines how many lines are displayed by the Evo system. There is little point in sending up more data than the display system can cope with, as the data will be thrown away by the display system in producing its images.

The `evoQualitySetting` can have the following values

evoQualitySetting	**Meaning**
0	The Evo system is displaying 32 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
1	The Evo system is displaying 64 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
2	The Evo system is displaying 128 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
3	The Evo system is displaying 256 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
4	The Evo system is displaying 512 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
5	The Evo system is displaying 1024 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
6	The Evo system is displaying 2048 lines of data, and the sonar will compress the data so that less than this number of lines is returned.
7	The Evo system is displaying 4096 lines of data, and the sonar will compress the data so that less than this number of lines is returned.

```
GEMX_SetGeminiEvoQuality(sonarID, 5);
```

GEMX_GetRequestedCompressionFactor

```
unsigned short GEMX_GetRequestedCompressionFactor(unsigned short
sonarID);
```

This function returns the compression factor applied when a ping was requested in `EvoC` mode. When running in `EvoC` mode, the compression factor is optimised for the Evo quality setting applied and the number of range lines requested. This function returns the actual compression factor used for the particular Evo quality setting and range requested.

The value returned by this function can have the following values.

Return Value	**Meaning**
1	The data will not have been compressed.
2	The data will have been compressed by a factor of 2 to 1.
4	The data will have been compressed by a factor of 4 to 1.
8	The data will have been compressed by a factor of 8 to 1.
16	The data will have been compressed by a factor of 16 to 1.

```
unsigned short compressionFactor = 0;
compressionFactor = GEMX_GetRequestedCompressionFactor(sonarID);
```

GEMX_SetPingMode

```
void GEMX_SetPingMode(unsigned short sonarID, unsigned short
pingMethod);
```

This function sets the Run_mode field in the ping configuration which will be sent to the head when a ping is requested.

pingMethod	**Meaning**
0	Ping once on receipt of ping configuration message
1	Ping repeatedly at interval fixed by GEMX_SetInterPingPeriod

```
GEMX_SetPingMode(sonarID, 1);
```

Default value: `pingMethod        0`

GEMX_SetExtModeOutOfWaterOverride

```
void GEMX_SetExtModeOutOfWaterOverride(unsigned short sonarID,
unsigned short outOfWaterOverride);
```

This function sets the Out of Water Override sub-field of the Ext_Mode field in the ping configuration which will be sent to the head when a ping is requested.

outOfWaterOverride	Meaning
0	Do not ping when out of water
1	Ping regardless of out of water indicator

```
GEMX_SetExtModeOutOfWaterOverride(sonarID, 0);
```

Default value: `outOfWaterOverride     0`

GEMX_SetInterPingPeriod

```
void GEMX_SetInterPingPeriod(unsigned short sonarID, unsigned int
periodInMicroSeconds);
```

This function sets the Inter_ping field in the ping configuration which will be sent to the head when a ping is requested. This field specifies the time delay between the start of one ping and the start of the next ping when the sonar head is pinging continuously, e.g. set using GEMX_SetPingMode.

```
GEMX_SetInterPingPeriod(sonarID, 200000);
```

Default value: `periodInMicroSeconds` 0
Minimum value:0
Maximum value:999 milliseconds

GEMX_AutoTXLength

```
unsigned short GEMX_AutoTXLength(unsigned short sonarID, float
range);
```

This function sets the `m_txLength` field in the ping configuration which will be sent to the head when a ping is requested. It sets the optimum transmit pulse length for a given range, and is provided as an alternative to the function GEMX_SetTXLength.

`range` is the range (in m) which the sonar is expected to return data in this ping.

This function returns the length of the transmit pulse in cycles…

```
unsigned short txLength;
txLength = GEMX_AutoTXLength(sonarID, 25.0);
```

Default value: `range`	`1.0`
Minimum value:	1.0
Maximum value:	150.0

GEMX_SetTXLength

```
void GEMX_SetTXLength(unsigned short sonarID, unsigned short
txLength);
```

This function sets the `m_txLength` field in the ping configuration which will be sent to the head when a ping is requested. This field specifies the length of the transmit pulse in cycles.

Large values for `txLength` should be avoided as these can potentially damage the sonar at high ping rates. Use GEMX_AutoTxLength instead to set the optimum transmit pulse length for a given range.

A value of zero for `txLength` has a special meaning to the code when it will stop the sonar's transmitter and velocimeter transmitting so the sonar emits no sonar power but receives as normal (for use in calibration, for example).

```
GEMX_SetTXLength(sonarID, 4);
```

Default value: `txLength`	4
Minimum value:	4
Maximum value:	128

GEMX_GetTXLength

```
unsigned short GEMX_GetTXLength(unsigned short sonarID);
```

This function returns the `m_txLength` field from the ping configuration which will be sent to the head when a ping is requested. This field specifies the length of the transmit pulse in cycles.

```
unsigned short txLength;
txLength = GEMX_GetTXLength(sonarID);
```

Default value: `txLength`	4
Minimum value:	4
Maximum value:	128

GEMX_SetVelocimeterMode

```
void GEMX_SetVelocimeterMode(unsigned short sonarID, unsigned short
gainMode, unsigned short outputMode);
```

For Gemini models fitted with a velocimeter, this function sets the Vel_mode field in the ping configuration which will be sent to the head when a ping is requested.

gainMode	Meaning
0	Use auto gain
1	Use manual gain

outputMode	Meaning
0	Use velocimeter calculated speed of sound
1	Use speed of sound specified in this ping configuration message

```
GEMX_SetVelocimeterMode(sonarID, 0, 0);
```

Default value:		
	`gainMode`	0
	`outputMode`	0

The gain mode functionality will not be implemented in the Gemini Sonar Head, as the head automatically calculates the gain required by the velocimeter. Thus the sonar will ignore any value set in the gain mode field and behave as though it was always set to zero.

GEMX_SetRangeCompression

```
void GEMX_SetRangeCompression(unsigned short sonarID, unsigned short
compressionLevel, unsigned short compressionType);
```

This function sets the Rng_comp field in the ping configuration which will be sent to the head when a ping is requested.

compressionLevel	Meaning (i.e. compressionFactor)
0	No range compression
1	2 * range compression
2	4 * range compression
3	8 * range compression
4	16 * range compression

compressionType	Meaning
0	Use average compression
1	Use peak compression

```
GEMX_SetRangeCompression(sonarID, 0, 1);
```

Default value: `compressionLevel` `0`
`compressionType` `1`

The compressionFactor that the compressionLevel sets can be queried by calling 'GEMX_GetRequestedCompressionFactor' (see earlier).

For Gemini 720im, Micron Gemini and MK2 Chirp operation, it is recommended that peak compressionType (=1) be used for better results.

The range compression is used by the DLL's `SeaNetC` mode to reduce the amount of data being passed from the library to SeaNet. In the `SeaNetC` mode, the library looks at the range requested in the ping and selects a suitable value for range compression so that no more than 1500 lines are passed to SeaNet. Normally if the range compression has been set, the library will remove the compression before passing the data to the calling program, in `SeaNetC` mode, the compression is not removed and the compressed data is passed to SeaNet.

The range compression is used by the DLL's `EvoC` mode to optimise the bandwidth used to transfer the data from the sonar to the surface computer, whilst considering the performance of the Evo display being used. In the `EvoC` mode, the library looks at the range requested in the ping and Evo quality setting that has been set; and then selects a suitable value for range compression so an optimum number lines are passed. Normally if the range compression has been set, the library will remove the compression before passing the data to the calling program, in `EvoC` mode, the compression is not removed and the compressed data is passed to Evo.

GEMX_SetRLEThreshold

```
void GEMX_SetRLEThreshold(unsigned short sonarID, unsigned short
threshold);
```

This function sets the Rle_threshold field in the ping configuration which will be sent to the head when a ping is requested. The library removes the run length encoding from the data before it is presented to the calling program, so the calling program does not need to be aware of the run length encoding algorithm used. Using run length encoding may reduce the bandwidth required to get the data from the Gemini sonar head to the hardware running the calling software. Values of 1 and 2 are reserved for use within the software, and so will be increased to 3 if selected.

threshold	Meaning
0	No run length encoding
1	Not available for use, will be increased to 3
2	Not available for use, will be increased to 3
3	
...	
255	Maximum run length encoding

```
GEMX_SetRLEThreshold(sonarID, 0);
```

Default value:	`threshold`	0
Minimum value:		0
Maximum value:		255

GEMX_SetPingToDefaults

```
void GEMX_SetPingToDefaults(unsigned short sonarID);
```

This function returns all the fields of the ping configuration which will be sent to the head when a ping is requested back to their default values.

```
GEMX_SetPingToDefaults(sonarID);
```

GEMX_SendGeminiStayAlive

```
void GEMX_SendGeminiStayAlive(unsigned short sonarID);
```

This function sends a stay-alive message to the Gemini sonar head. If the head does not receive any communications data from the surface PC for 3 seconds it will stop sending data back to the surface PC. The head will stop pinging if it is in continuous ping mode until further communications are received. The stay-alive message is a simple way of keeping the communications alive between surface PC and the head.

The Gemini sonar head will respond with an acknowledge message which will result in a CGemAcknowledge packet being passed to the data handler callback function (with the type identifier of GEM_ACKNOWLEDGE). This packet contains timestamp information which may or may not be useful to the calling application.

```
GEMX_SendGeminiStayAlive(sonarID);
```

GEMX_SendGeminiPingConfig

```
void GEMX_SendGeminiPingConfig(unsigned short sonarID);
```

This function sends a ping configuration message to the Gemini sonar head, which will cause the head to issue one or more pings (depending on how the ping configuration has been set).

```
GEMX_SendGeminiPingConfig(sonarID);
```

GEMX_RebootSonar

```
void GEMX_RebootSonar(unsigned short sonarID);
```

This function issues a command to the Gemini sonar head to command it to reboot into the main software image in the Sonar head.

```
GEMX_RebootSonar(sonarID);
```

GEMX_SetVDSLSetting

```
void GEMX_SetVDSLSetting(unsigned short sonarID, unsigned short
level);
```

This function only applies to Gemini MK1/MK2 models which are fitted with VDSL communications.

The link between the sonar head and the surface is an ethernet link. In some configurations, the link may be carried over VDSL for part of its journey. In order to optimise the speed of the VDSL connection depending on the amount of electrical noise the link was subjected to, three different settings are provided. This function allows the calling program to change the settings of the VDSL link and then retrigger a VDSL rate adaption to make use of the new settings.

`level` signifies which set of VDSL settings to use, as defined below.

level	Meaning
0	Normal electrical noise environment
1	Medium electrical noise environment
2	High electrical noise environment

```
GEMX_SetVDSLSetting(sonarID, 0);
```

Calling this function will cause a VDSL rate adaption, which will interrupt communications with the sonar head for a short period.

GEMX_GetDeviceName

```
void GEMX_GetDeviceName (unsigned short sonarID, char* pBuf,
unsigned int len)
```

Copies a null-terminated ASCII string name for the device into `pBuf`. A maximum of `len` characters (including the `NULL` string terminator) are copied.

If `len` is less than 10, a short name is returned e.g. `"720i"`. Otherwise a full name e.g. `"Gemini Imager 720i"` is returned. A minimum of 32 characters are recommended for requesting the full name.

```
char buf[32];
GEMX_GetDeviceName(sonarID, buf, 32);
```

GEMX_GetGeminiFrequency

```
int GEMX_GetGeminiFrequency(unsigned short sonarID)
```

Returns the operating frequency of the sonar in Hz.

```
int freq = GEMX_GetGeminiFrequency(sonarID);
```

GEMX_GetGeminiModFrequency

```
int GEMX_GetGeminiModFrequency(unsigned short sonarID)
```

Returns the modulation frequency of the sonar in Hz.

In conjunction with the speed of sound, the modulation frequency can be used to calculate the size of one range line in metres using the following formula:

[Range line in metres] = ([speed of sound] / 2) / [Modulation Frequency]

```
int modFreq = GEMX_GetGeminiModFrequency(sonarID);
```

GEMX_SetGeminiBeams

```
void GEMX_SetGeminiBeams(unsigned short sonarID, unsigned short
nBeams)
```

Only supported for the MK2 platform.

Sets the number of beams (`nBeams`) that the Gemini will form. Default is set to 256 beams for MK2 products except 1200ikd. Default for 1200ikd is 512 beams

Generate a bearing table based on nBeams (256 or 512) and send to the sonar.

```
GEMX_SetGeminiBeams(sonarID, 512); // 512 for MK2 else 256 beams
```

GEMX_GetGeminiBeams

```
int GEMX_GetGeminiBeams(unsigned short sonarID, float *pBrgTbl)
```

Returns the number of beams (`nBeams`) that the Gemini will form.

`pBrgTbl` is a pointer to memory (allocated by the calling function) where a table of `nBeams` float values will be written by GEMX_GetGeminiBeams. If this is not required, set `pBrgTbl` to `NULL`.

If `pBrgTbl` is not `NULL`, then GEMX_GetGeminiBeams will write a table of the bearings in radians for each beam at this address. The calling function must allocate memory for nBeams float values and pass a pointer to that memory to GEMX_GetGeminiBeams. A bearing of zero corresponds to the centre of the Gemini's scan range (boresight).

Example usage:

```
int nBeams = GEMX_GetGeminiBeams(sonarID, NULL);
float* pBrgTbl = (float*)malloc(nBeams * sizeof(float));
if (pBrgTbl != NULL)
{
  GEMX_GetGeminiBeams(sonarID, pBrgTbl);
  ... use the bearing data ...
  free(pBrgTbl);
}
```

GEMX_ConfigureChirpMode

```
void GEMX_ConfigureChirpMode(unsigned short sonarID, CHIRP_MODE
chirpMode );
```

Only supported for Mk2, 720im and MicronGemini

Please see structure below for possible “chirp mode“ values.

```
enum CHIRP_MODE{
   CHIRP_DISABLED,
   CHIRP_ENABLED,
   CHIRP_AUTO
};
```

This function allows ability to switch between continuous wave or chirp mode. Default is the auto mode. User can also force to switch between Chirp On/Off.

```
GEMX_ConfigureChirpMode(sonarID, chirpMode);
```

Example usage:

```
// To enable Chirp in auto mode
GEMX_ConfigureChirpMode(sonarID, CHIRP_AUTO);

// To enable Chirp mode
GEMX_ConfigureChirpMode(sonarID, CHIRP_ENABLED);

// To disable Chirp mode
GEMX_ConfigureChirpMode(sonarID, CHIRP_DISABLED);
```

GEMX_GetActiveChirpMode

```
bool GEMX_GetActiveChirpMode (unsigned short sonarID);
```

Only supported for Mk2, 720im and MicronGemini

This function returns the active chirp mode. True if chirp is enabled otherwise false.

```
Bool chirpEnable = GEMX_GetActiveChirpMode(sonarID);
```

GEMX_ConfigureMainPort

```
bool GEMX_ConfigureMainPort(unsigned short sonarID, bool negotiate,
bool rs232, unsigned int baudRate, const char* portName);
```

Only supported for Micro platforms (720im and MicronGemini).

The GeminiSDK only support maximum of 16xMicro platforms on the regular COM port. To open the first Gemini Micro platform on regular COM port use SonarID 0xFFFF, to open second Gemini Micro platform on regular COM port use SonarID 0xFFFE and so on. Once the port is opened then sonar will start sending messages with the correct sonar ID. For further configuration, use the actual sonarID. See example below for multiple Gemini Micro platforms on regular COM port

For Linux using regular COM port: Add user in the same group of permission to open the device e.g. /dev/ttyUSB0". To change the permission of the device run the command with super user permission "sudo usermod -a -G dialout MY_USER_NAME"

For Linux using Tritech USB to Serial adaptor: Run the "Tritech-USB-Serial.sh" with super user permission then run the SDK application. This script only required to run once on first installation before running the application.

This function sets the main port to either RS232 (Full duplex) or RS485 (Half duplex) mode based on the 'rs232' boolean parameter provided in the API. In addition, further parameters set a negotiation function which on startup will adapt the rate to the quality of the line, starting at 115,200 baud and stepping up to 921,600 baud if the line quality handles that rate without dropped packets. When 'negotiate' is set false, the 'baudRate' parameter will be read to manually set the rate (otherwise baudRate value will be ignored when negotiate is true). Finally, the 'portName' parameter sets the computer COM Port to which the Gemini Micro platform is connected; NULL = Tritech USB to Serial adaptor being used, COM 'X' = Gemini Micro platform connected to another computer COM Port (where 'X' = its number). The function will return a value true meaning success else false if it failed to configure the port connection. Note, user will only be able change main port mode if running on the Gemini Micro platform serial interface.
Important: To close a currently opened port, pass in "None" as the portName.

Note: Negotiation is limited to 921600 for 720im and MicronGemini (RS232). Micron Gemini can be negotiated up to 4000000 in RS485 mode.

Example usage:

// Configure main port in RS232 mode @ 230,400 baud using Tritech USB to Serial adaptor
```
Int retVal = GEMX_ConfigureMainPort(sonarID, false, true, 230400,
NULL);
```

// Configure main port in RS485 mode with negotiation using Tritech USB to Serial adaptor
```
Int retVal = GEMX_ConfigureMainPort(sonarID, true, false, 921600,
NULL);
```

// Configure main port in RS232 mode with negotiation using computer COM Port 1
```
Int retVal = GEMX_ConfigureMainPort(65535, true, true, 115200,
“COM1”);
```

Example of Multiple Gemini Micro platforms on regular COM ports

// In Linux, open 2x Micro Gemini Micro platforms connected on /dev/ttyUSB0 port in RS232 mode with negotiation on
```
GEMX_ConfigureMainPort(65535, true, true, 115200, “/dev/ttyUSB0”);
GEMX_ConfigureMainPort(65534, true, true, 115200, “/dev/ttyUSB1”);
```

After a successful port open, The GeminiSDK will start sending status message with the correct sonar ID.

Following example is configuring the fix baudrate with the correct sonar ID.
```
Int retVal = GEMX_ConfigureMainPort(901, false, true, 230400,
“/dev/ttyUSB0”);
```

GEMX_ConfigureAutoRangeFrequency

```
bool GEMX_ConfigureAutoRangeFrequency(unsigned short
sonarID, RangeFrequencyConfig autoRangeConfig);
```

Only supported for 1200ikd & 1200id

```
enum FREQUENCY{
   FREQUENCY_AUTO, // Switch between low/high resolution based on the range threshold
specified
   FREQUENCY_LOW,  // Force to run with low frequency 720
   FREQUENCY_HIGH  // Force to run with high frequency 1200
};

struct RangeFrequencyConfig{
   RangeFrequencyConfig()
   : m_frequency( FREQUENCY_AUTO ) // default: resolution mode
   , m_rangeThreshold( 40.0 ) // only used in auto mode
   {
   }
   FREQUENCY   m_frequency;
   Double        m_rangeThreshold;   // Configure the range threshold to switch between
                                      //  low / high resolution. Range (1-50)m
};
```

This function configures the GeminiComms to switch sonar frequency based on range selected by the user or user can force to switch between lower frequency (723kHz) or higher frequency (1.2MHz).

Higher frequency can only be applied on ranges 0 to 50m.

Example usage:

// Configure 1200ikd to switch frequency based on the range thread hold e.g. sonar will run in the higher frequency if range is selected 10m or below

```
autoRangeConfig.m_frequency = FREQUENCY_AUTO;
autoRangeConfig. m_rangeThreshold = 10.0
GEMX_ConfigureAutoRangeFrequency(sonarID, autoRangeConfig);
```

// Configure 1200ikd to configure lower frequency

```
autoRangeConfig.m_frequency = FREQUENCY_LOW;
GEMX_ConfigureAutoRangeFrequency(sonarID, autoRangeConfig);
```

// Configure 1200ikd to configure higher frequency

```
autoRangeConfig.m_frequency = FREQUENCY_HIGH;
GEMX_ConfigureAutoRangeFrequency(sonarID, autoRangeConfig);
```

GEMX_AutoHighRangeResolution

```
bool GEMX_ConfigureAutoHighRangeFrequency(unsigned short
sonarID, bool enable);
```

Only supported for MK2 products with firmware version (BF: 0x200e, DA: 0x2008 or above)

This function configures the GeminiComms to switch between high/low range resolutions based on the range selected by the user. Enabling this feature would result in double the range lines for a given range, decrease the range line separation and increasing the range resolution.

GEMX_SetNoiseReduction

```
bool GEMX_SetNoiseReduction (unsigned short          sonarID, bool
enable);
```

Only supported for 720im and MicronGemini

This function configures the GeminiComms to enable / disable Noise Reduction filter in the sonar. Enabling this feature would result in removing the noise from the image.

GEMX_720imEnableH264

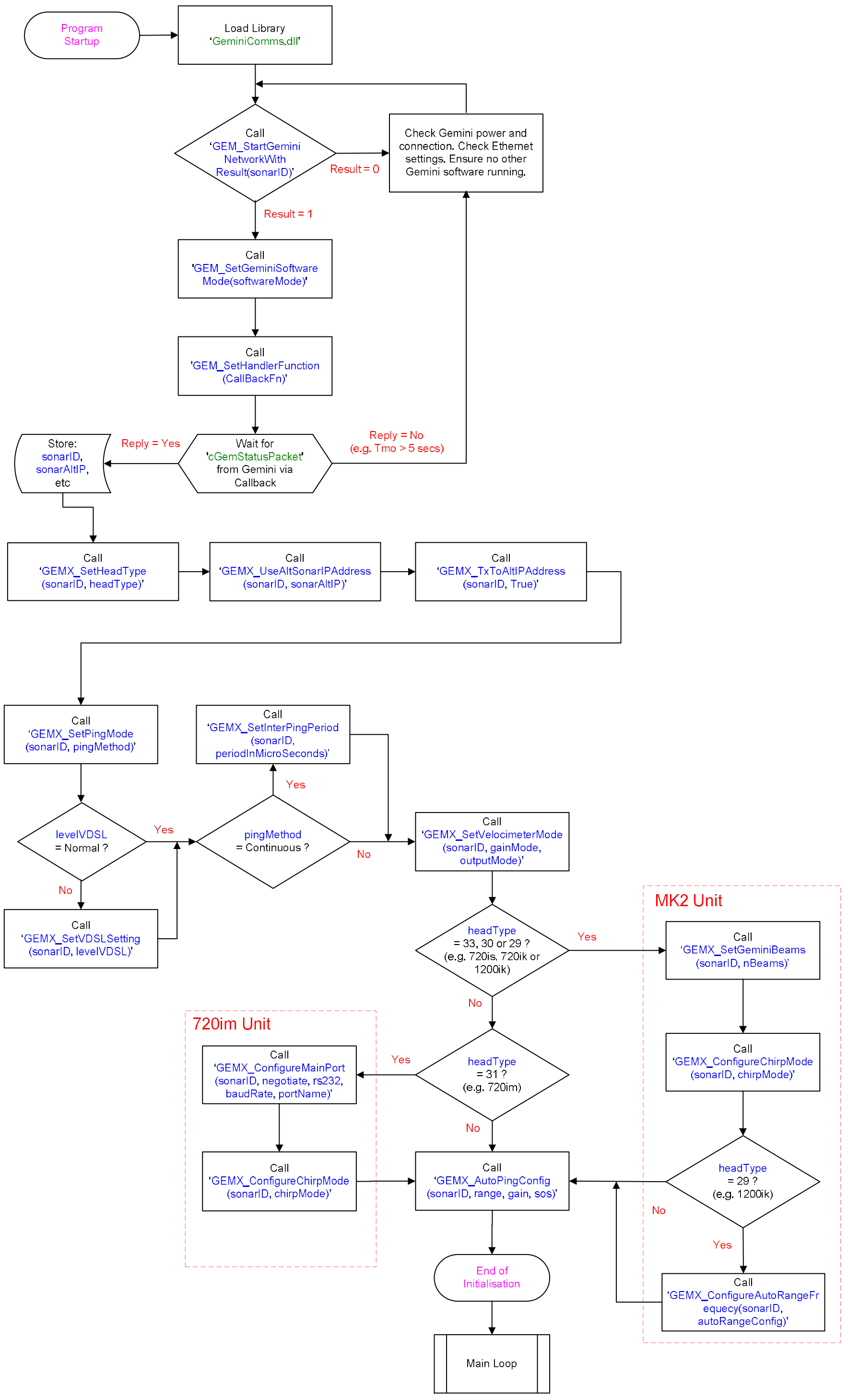
```
void GEMX_720imEnableH264 (unsigned short sonarID, bool enableH264);
```

Only supported for 720im and MicronGemini

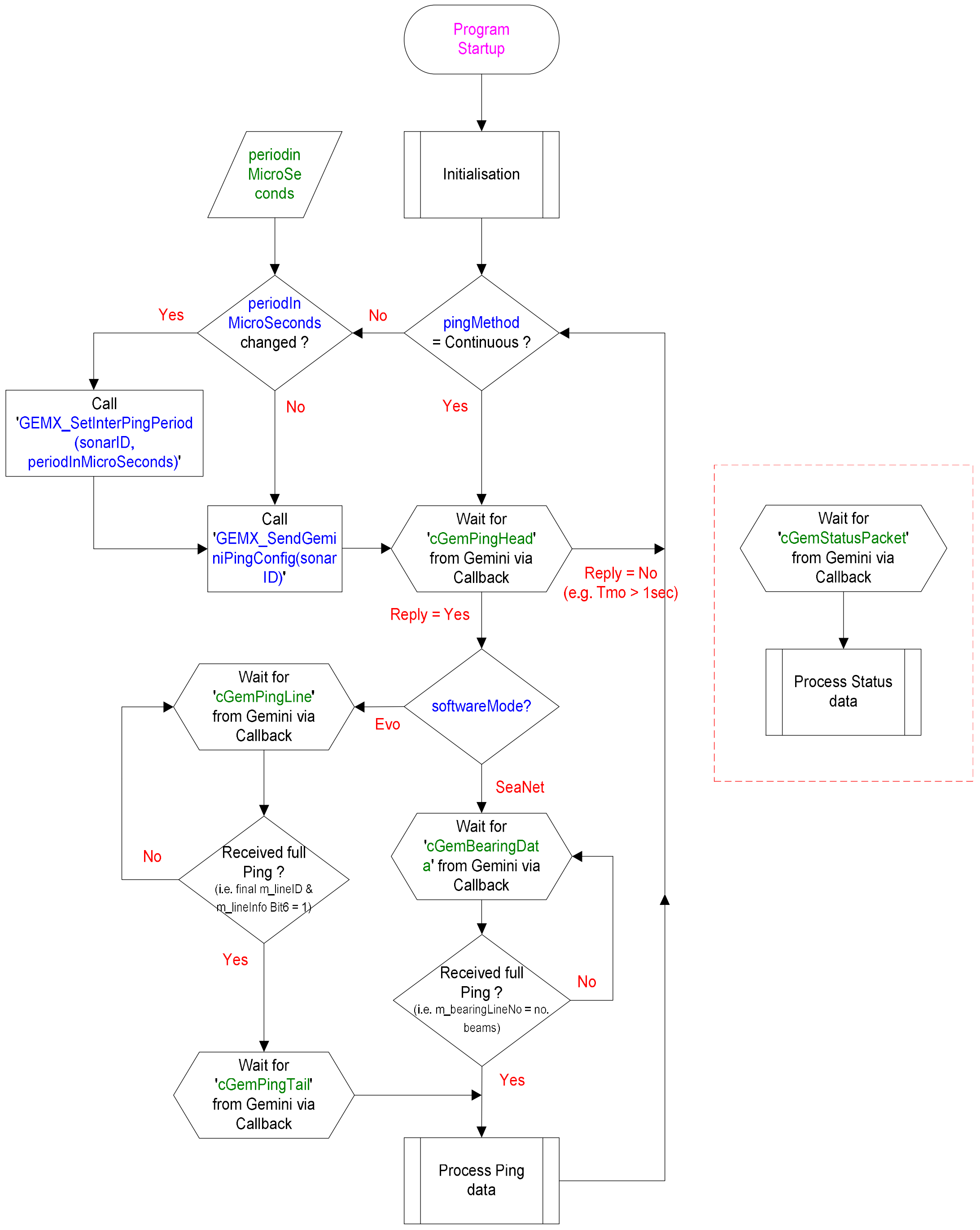
This function configures the GeminiComms to enable / disable H264 Compression for Ethernet connections. H264 compression is enabled by default for Serial (RS-485/232) communications.

Appendix A

Flow Chart – Basic Initialisation routine


Program Startup
Load Library 'GeminiComms.dll'
Call 'GEM_StartGeminiNetworkWithResult(sonarID)'
Result = 0
Check Gemini power and connection. Check Ethernet settings. Ensure no other Gemini software running.
Result = 1
Call 'GEM_SetGeminiSoftwareMode(softwareMode)'
Call 'GEM_SetHandlerFunction(CallBackFn)'
Wait for 'cGemStatusPacket' from Gemini via Callback
Reply = No (e.g. Tmo > 5 secs)
Reply = Yes
Store: sonarID, sonarAltIP, etc
Call 'GEMX_SetHeadType(sonarID, headType)'
Call 'GEMX_UseAltSonarIPAddress(sonarID, sonarAltIP)'
Call 'GEMX_TxToAltIPAddress(sonarID, True)'
Call 'GEMX_SetPingMode(sonarID, pingMethod)'
levelVDSL = Normal ?
No
Call 'GEMX_SetVDSLSetting(sonarID, levelVDSL)'
Yes
pingMethod = Continuous ?
Yes
Call 'GEMX_SetInterPingPeriod(sonarID, periodInMicroSeconds)'
No
Call 'GEMX_SetVelocimeterMode(sonarID, gainMode, outputMode)'
headType = 33, 30 or 29 ? (e.g. 720is, 720ik or 1200ik)
Yes
MK2 Unit
Call 'GEMX_SetGeminiBeams(sonarID, nBeams)'
Call 'GEMX_ConfigureChirpMode(sonarID, chirpMode)'
headType = 29 ? (e.g. 1200ik)
Yes
Call 'GEMX_ConfigureAutoRangeFrequency(sonarID, autoRangeConfig)'
No
headType = 31 ? (e.g. 720im)
Yes
720im Unit
Call 'GEMX_ConfigureMainPort(sonarID, negotiate, rs232, baudRate, portName)'
Call 'GEMX_ConfigureChirpMode(sonarID, chirpMode)'
No
Call 'GEMX_AutoPingConfig(sonarID, range, gain, sos)'
End of Initialisation
Main Loop

Flow Chart – Main Loop routine



Program Startup
Initialisation
periodin MicroSeconds
periodIn MicroSeconds changed ?
pingMethod = Continuous ?
Yes
No
No
Yes
Call 'GEMX_SetInterPingPeriod (sonarID, periodInMicroSeconds)'
Call 'GEMX_SendGeminiPingConfig(sonarID)'
Wait for 'cGemPingHead' from Gemini via Callback
Reply = No (e.g. Tmo > 1sec)
Reply = Yes
softwareMode?
Evo
SeaNet
Wait for 'cGemPingLine' from Gemini via Callback
Received full Ping ? (i.e. final m_lineID & m_lineInfo Bit6 = 1)
No
Yes
Wait for 'cGemPingTail' from Gemini via Callback
Wait for 'cGemBearingData' from Gemini via Callback
Received full Ping ? (i.e. m_bearingLineNo = no. beams)
No
Yes
Process Ping data
Wait for 'cGemStatusPacket' from Gemini via Callback
Process Status data

Appendix B

Plotting the Data

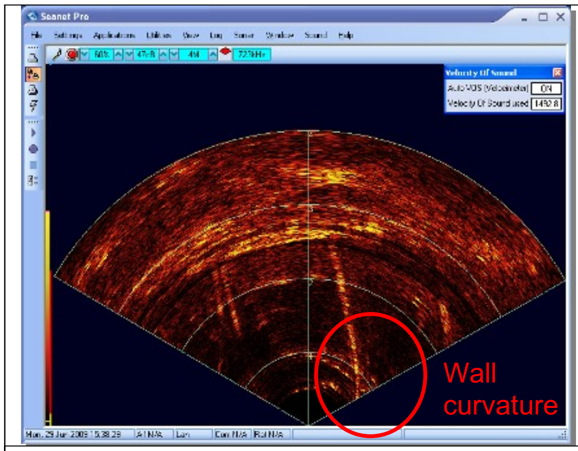
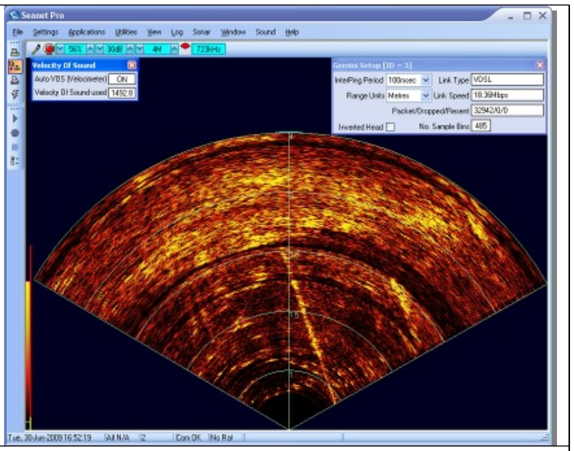
The Gemini sonar field of view is described by four attributes: aperture, number of beams, beam spacing pattern, and range.

Range is set when commanding a ping. The functions GEMX_GetGeminiAperture and GEMX_GetGeminiBeams can be used to retrieve the other three attributes.

```
int aperture = GEMX_GetGeminiAperture (sonarID);
int nBeams = GEMX_GetGeminiBeams(sonarID, NULL);
float* pBrgTbl = (float*)malloc(nBeams * sizeof(float));
if (pBrgTbl != NULL)
{
  GEMX_GetGeminiBeams(sonarID, pBrgTbl);
  ... use the bearing data ...
  free(pBrgTbl);
}
```

In the `Evo`/`EvoC` modes each line of data returned consists of `nBeams` values representing the image over `aperture` degrees at a particular range. In the `SeaNet`/`SeaNetC` modes, there are `nBeams` bearing lines returned representing the image data along a particular bearing.

The importance of using the bearing table (`pBrgTbl`) information is demonstrated in the two images below. In the first image, an (incorrect) bearing table has been used, resulting in false curvature of the test tank wall. In the second image, the correct bearing table from the library is used, resulting in the correct straight wall of the test tank.

	
Incorrect bearing table	Correct bearing table from library

Appendix C

MicronGemini equation

```
/*Converts the unsigned value from the sonar to a temperature in degrees */
Double MS5837_Temp_degC(unsigned short temp_val)
{
	if (temp_val != 0xFFFF)
		return (double)((short)(temp_val) - 2000) / 100.0;
	else
		return 0.0;
}

#define MS5837_PRES_RANGE   300000.0
/*Returns the 16-bit representation back to a abs. pressure in millibar*/
double MS5837_Pres_bar(unsigned short pres_val)
{
	if (pres_val != 0xFFFF)
		return (double)((MS5837_PRES_RANGE/pow(2.0, 16.0)) * (double)pres_val) / 
10000;
	else
		return 0.0;
}

/*Derives a depth in m, taken from the SK Bathy 700 Manual 0367-SOM-00001-04*/
double CalcDepth_m(double abs_press_bar, double atmos_press_bar, double density_kg_l,
                                                           double loc_grav_ms2)
{
	double A1, A2, gn, P, a, g, p, depth;

	if(abs_press_bar == 0.0)
		return 0.0;

	A1 = 0.70307;
	A2 = 0.01450377;

	gn = 9.80665; //Std Gravity m/s2

	P = abs_press_bar * 14.50377;     //Abs pressure in psi
	a = atmos_press_bar * 1000.0;
	g = loc_grav_ms2;
	p = density_kg_l;

	depth = (gn / g) * ((A1 * (P - (A2 * a))) / p);

	return depth;
}
```

Note: Specifications for the sensor are as follows (if a measurement falls outside these values the value will be reported as 0xFFFF):

- Temp = -20/+85oC (resolution typ. 0.002, abs accuracy is +/- 4oC although I suspect in real life it's a lot less than this)
- Pressure – 0-30bar (resolution typ. 0.28mbar, abs accuracy is +/- 400mbar)