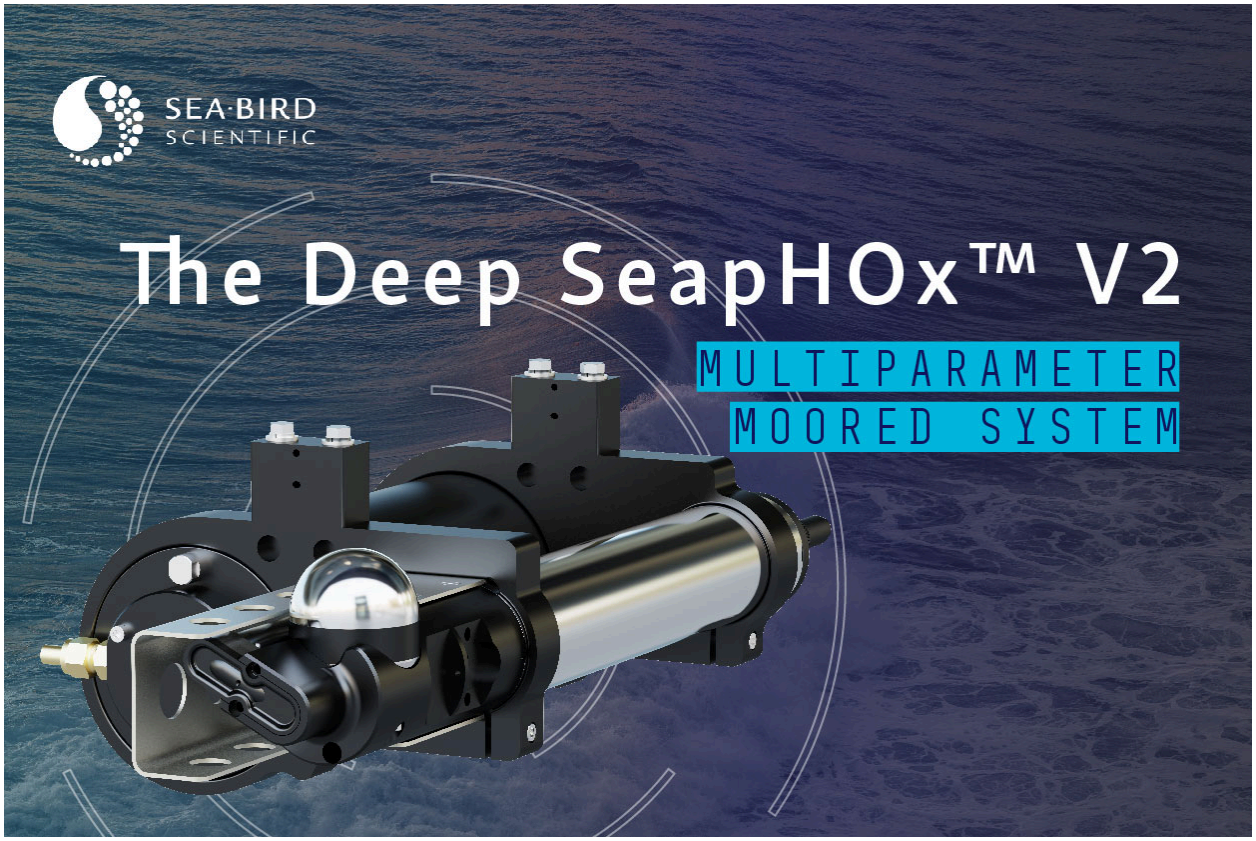
PHSEN Series H

Data & Development Requirements

For Sea-Bird SeapHOx pH Unit



Vardaro, M., Manalang, D., Rosburg, K.C., Wingard, C. and Ruef, W.

Created:
11 April 2025

Table of Contents

Change Log

Date	Changes	Editor
11 Apr. 2025	Document created	M. Vardaro
09 Sep 2025	Revisions to ph_total code	M. Vardaro
05 Dec 2025	Revising to correct Deep/Shallow errors	M. Vardaro

I. Introduction

As outlined in ECR-1204 and ECR-1318 (approved at the System Level in April and November 2024, respectively), the PHSEN-A/D instruments (Sunburst SAMI-pH) have been replaced by PHSEN-G/H series (Sea-Bird SeapHOx V2) instruments. Based on testing and documentation, a new driver and some algorithm edits are necessary. Details on Preload and Asset Management, ion_function algorithms, and Port Agent/Driver updates are below.

Major Differences

This section lays out the major operational differences between the existing Sunburst (PHSEN-A/D) and the Sea-Bird SeapHOx (PHSEN-G/H).

The Sunburst Sensor Sami II pH instrument allows the user to measure seawater pH on the total scale (pHT) using a colorimetric reaction occurring in an indicator solution. A seawater sample is pumped through the instrument and injected with a pH-dependent indicator solution. Two wavelength-specific LEDs send alternating pulses of light through the indicator-sample mixture as it is pumped through a flow cell. The ratio of the absorbance at the two wavelengths with and without the dye, Beer's law, and the molar absorptivities of the indicator are then used to calculate the concentration of protonated and un-protonated indicator. The indicator pK'a is then used to calculate pH. pHT measures the ratio of indicator dye in a seawater sample and infers the activity of both protons (free hydrogen ions) and hydrogen sulfate ions in the colorimetric reaction. The use of reagent limits the sampling frequency and overall shelf life of the instrument; we have also had frequent housing leaks and data quality issues.

The Deep SeapHOx V2 sensor uses ion-sensitive field effect transistor (ISFET) technology to measure pH in marine environments at depths to 2000 meters. The sensor has internal memory and an internal battery pack so that it can operate autonomously for a long-term deployment. When an SBE-37-SMP-ODO is attached to the Deep SeaFET, the system operates in Deep SeapHOx mode, which also measures salinity, temperature, depth, and oxygen, and has an internal pump. The Deep SeapHOx uses a manufacturer-attached "Y" cable to connect the Deep SeaFET and SBE-37-SMP-ODO. The manufacturer-supplied software lets the user set up the sensor, monitor graphical data in real time, upload stored data, and process that data. During the deployment of a Deep SeapHOx, the Deep SeaFET queries the SBE-37-SMP-ODO to collect data, which is saved to the Deep SeaFET. No data is stored in the SBE-37-SMP-ODO. The end flange cap is used with flow cells when the Deep SeaFET is connected to the SBE-37-SMP-ODO.

The primary sensor element of the Deep SeapHOx V2 is a special deep-packaged ISFET, a solid-state sensor that senses pH in marine environments. Unlike the SeaFET and shallow SeapHOx, there is no internal reference electrode on the Deep SeapHOx. The external reference electrode has a Ag/AgCl reference electrode in direct contact with seawater. The potential of this electrode varies with pH and chloride concentration, so unless the chloride concentration is known, the external reference is not stable. To correct this, salinity is used as

an approximation of chloride concentration. The salinity data that is calculated by the SBE-37-SMP-ODO is applied to the pH external data and significantly reduces measurement errors, and gives the most accurate and stable pH data.

II. Preload & Asset Management

Calibrations

The following calibration values shall be included in the asset management calibration data (see examples here: https://github.com/oceanobservatories/asset-management/tree/master/calibration/PHSENH)

`k0`	K0 Coefficient
`k2`	K2 Coefficient
`f1, f2, f3, f4, f5, f6`	f(P) coefficients
`a0, a1, a2, a3`	Temperature from counts
`ptempa0, ptempa1, ptempa2, ptca0, ptca1, ptca2, ptcb0, ptcb1, ptcb2, pa0, pa1, pa2`	Pressure from counts
`wbotc, g, h, i, j, ctcor, cpcor`	Conductivity from counts
`c0, c1, c2, coeff_e, a0, a1, a2, b0, b1, therm_ta0, therm_ta1, therm_ta2, therm_ta3`	Dissolved oxygen coefficients

Deployment

The following values shall be included in the asset management deployment data.

`CC_latitude`	Sensor latitude from Deployment
`CC_longitude`	Sensor longitude from Deployment
`CC_depth`	Sensor depth from Deployment

Parameter Definitions

The following parameters need to be added to ParameterDefs.csv in the preload repository. New parameters are broken into three categories: 1) array type parameters, 2) quantity/scalar type parameters, 3) functions type parameters that will need to be linked to a new ion-function in the ParameterFunctions.csv and for which new ion-functions must be written.

The following table lists necessary new array parameters.

Parameter	Description	Why New is Needed
N/A	N/A	N/A

The following table lists necessary new quantity parameters (i.e. scalar values).

Parameter	Description	Why New is Needed
framesync	Framesync	Doesn't exist
event_flags	Bit-flags in hex format 0000	Doesn't exist
temperature_counts	Temp. raw ADC counts	Doesn't exist
ph_external_reference_voltage_counts	pH ref. Voltage raw ADC counts	Doesn't exist
ph_voltage_counts	pH counter voltage raw ADC counts	Doesn't exist
ph_current_counts	pH base current raw ADC counts	Doesn't exist
ph_counter_current_counts	pH counter current raw ADC counts	Doesn't exist
pressure_counts	Raw pressure ADC counts	Doesn't exist
pressure_temperature_counts	Raw pressure-temperature ADC counts	Doesn't exist
conductivity_frequency	Conductivity frequency (Hz)	Doesn't exist
oxygen_phase_delay	Oxygen phase delay (microsec)	Doesn't exist
oxygen_thermistor_voltage	O2 thermistor voltage (V)	Doesn't exist
internal_temperature_counts	Internal temperature code counts	Doesn't exist
internal_humidity_counts	Internal humidity code counts	Doesn't exist

The following table lists necessary new function parameters. These will need associated new ion-functions to be written and will need to be linked to the ion-function in preload.

Parameter	Description	Why New is Needed
ph_total	Total pH from the SeapHOx sensor calculated from the external voltage (vrs_ext), temperature (degC), salinity (psu), pressure (dbar), and the calibration coefficients (k0, k2, f).	Doesn't exist

Streams

The following streams need to be defined in ParameterDictionary.csv in the preload repository.

phsen_h_format0	

phsen_h_format0

This is the stream PHSENH configuration data and includes the following parameters:

- time
- port_timestamp
- driver_timestamp
- internal_timestamp
- preferred_timestamp
- ingestion_timestamp
- framesync
- event_flags
- temperature_counts
- ph_external_reference_voltage_counts
- ph_voltage_counts
- ph_current_counts
- ph_counter_current_counts
- pressure_counts
- pressure_temperature_counts
- conductivity_frequency
- oxygen_phase_delay
- oxygen_thermistor_voltage
- internal_temperature_counts
- internal_humidity_counts
- dissolved_oxygen - **PD3777**
- ph_total (**NEW**)
- sea_water_temperature - **PD4**
- sea_water_practical_salinity - **PD3**

- `sea_water_pressure` - **PD2**
- `sea_water_electrical_conductivity` - **PD1**
- `sea_water_density` - **PD5**
- `depth_from_pressure` - **PD8084**
- `Internal_temperature` - **PD8203**
- `Internal_humidity` - **PD8204**

Function Definitions

The following ion-function definitions need to be added to ParameterFunctions.csv in the preload repository. This section is a bit redundant, but serves as a reminder that new ion-functions must be added to preload.

Function	Inputs	Description

III. Ion Functions

The following functions shall be coded into the ion-function repository.

Example code for the SeapHOx Ion Function can be found here: https://github.com/Sea-BirdScientific/seabirdscientific/blob/main/src/seabirdscientific

and here:

https://bitbucket.org/ooicgsn/cgsn-processing/src/master/cgsn_processing/process/proc_cphox.py

Many of the functions could be adapted from the ion_functions for existing Sea-Bird CTD sensors: https://github.com/ooici/ion-functions/blob/master/ion_functions/data/ctd_functions.py

Conversions

```
# Native imports
from math import e, floor
from typing import Literal

# Third-party imports
import gsw
import numpy as np
from scipy import stats

# Sea-Bird imports

# Internal imports
from .cal_coefficients import (
    ConductivityCoefficients,
    Oxygen43Coefficients,
    Oxygen63Coefficients,
    PARCoefficients,
    PH18Coefficients,
    PHSeaFETInternalCoefficients,
    PHSeaFETExternalCoefficients,
    PressureCoefficients,
    TemperatureCoefficients,
    Thermistor63Coefficients,
    ECOCoefficients,
)


DBAR_TO_PSI = 1.450377
PSI_TO_DBAR = 0.6894759
```

```
OXYGEN_PHASE_TO_VOLTS = 39.457071
KELVIN_OFFSET_0C = 273.15
KELVIN_OFFSET_25C = 298.15
OXYGEN_MLPERL_TO_MGPERL = 1.42903
OXYGEN_MLPERL_TO_UMOLPERKG = 44660
# taken from
https://blog.seabird.com/ufaqs/what-is-the-difference-in-temperature-expressio
ns-between-ipts-68-and-its-90/ # pylint: disable=line-too-long
ITS90_TO_IPTS68 = 1.00024
# micro moles of nitrate to milligrams of nitrogen per liter
UMNO3_TO_MGNL = 0.014007
# [J K^{-1} mol^{-1}] Gas constant, NIST Reference on Constants retrieved
10-05-2015
R = 8.3144621
# [Coulombs mol^{-1}] Faraday constant, NIST Reference on Constants retrieved
10-05-2015
F = 96485.365

def convert_ph_voltage_counts(ph_counts: np.ndarray):
    """Convert pH voltage counts to a floating point value

    :param ph_counts: pH voltage counts
    :return: pH voltage
    """
    adc_vref = 2.5
    gain = 1
    adc_23bit = 8388608
    ph_volts = adc_vref / gain * (ph_counts / adc_23bit - 1)
    return ph_volts
```

Temperature

```
def convert_temperature(
    temperature_counts_in: np.ndarray,
    coefs: TemperatureCoefficients,
    standard: Literal["ITS90", "IPTS68"] = "ITS90",
    units: Literal["C", "F"] = "C",
    use_mv_r: bool = False,
):
    """Returns the value after converting it to degrees C, ITS-90.

    Data is expected to be raw data from an instrument in A/D counts

    :param temperature_counts_in: temperature value to convert in A/D
        counts
    :param coefs: calibration coefficients for the temperature sensor
    :param standard: whether to use ITS90 or to use IPTS-68 calibration
```

```
        standard
    :param units: whether to use celsius or to convert to fahrenheit
    :param use_mv_r: true to perform extra conversion steps required by
        some instruments (check the cal sheet to see if this is required)

    :return: temperature val converted to ITS-90 degrees C
    """

    if use_mv_r:
        mv = (temperature_counts_in - 524288) / 1.6e007
        r = (mv * 2.900e009 + 1.024e008) / (2.048e004 - mv * 2.0e005)
        temperature_counts = r
    else:
        temperature_counts = temperature_counts_in

    log_t = np.log(temperature_counts)
    temperature = (
        1 / (coefs.a0 + coefs.a1 * log_t + coefs.a2 * log_t**2 + coefs.a3 *
log_t**3)
    ) - KELVIN_OFFSET_0C

    if standard == "IPTS68":
        temperature *= ITS90_TO_IPTS68
    if units == "F":
        temperature = temperature * 9 / 5 + 32  # Convert C to F

    return temperature
```

Pressure

```
def convert_pressure(
    pressure_count: np.ndarray,
    compensation_voltage: np.ndarray,
    coefs: PressureCoefficients,
    units: Literal["dbar", "psia"] = "psia",
):
    """Converts pressure counts to PSIA (pounds per square inch, abolute).

    pressure_count and compensation_voltage are expected to be raw data
    from an instrument in A/D counts

    :param pressure_count: pressure value to convert, in A/D counts
    :param compensation_voltage: pressure temperature compensation
        voltage, in counts or volts depending on the instrument
    :param coefs: calibration coefficients for the pressure sensor
    :param units: whether or not to use psia or dbar as the returned
        unit type
```

```
    :return: pressure val in PSIA
    """
    sea_level_pressure = 14.7

    t = (
        coefs.ptempa0
        + coefs.ptempa1 * compensation_voltage
        + coefs.ptempa2 * compensation_voltage**2
    )
    x = pressure_count - coefs.ptca0 - coefs.ptca1 * t - coefs.ptca2 * t**2
    n = x * coefs.ptcb0 / (coefs.ptcb0 + coefs.ptcb1 * t + coefs.ptcb2 * t**2)
    pressure = coefs.pa0 + coefs.pa1 * n + coefs.pa2 * n**2 -
sea_level_pressure

    if units == "dbar":
        pressure *= PSI_TO_DBAR

    return pressure
```

Conductivity

```
def convert_conductivity(
    conductivity_count: np.ndarray,
    temperature: np.ndarray,
    pressure: np.ndarray,
    coefs: ConductivityCoefficients,
):
    """Converts raw conductivity counts to S/m.

    Data is expected to be raw data from instrument in A/D counts

    :param conductivity_count: conductivity value to convert, in A/D
        counts
    :param temperature: reference temperature, in degrees C
    :param pressure: reference pressure, in dbar
    :param coefs: calibration coefficient for the conductivity sensor

    :return: conductivity val converted to S/m
    """
    f = conductivity_count * np.sqrt(1 + coefs.wbotc * temperature) / 1000
    numerator = coefs.g + coefs.h * f**2 + coefs.i * f**3 + coefs.j * f**4
    denominator = 1 + coefs.ctcor * temperature + coefs.cpcor * pressure
    return numerator / denominator
```

Depth from Pressure

```
def depth_from_pressure(
    pressure_in: np.ndarray,
    latitude: float,
    depth_units: Literal["m", "ft"] = "m",
    pressure_units: Literal["dbar", "psi"] = "dbar",
):
    """Derive depth from pressure and latitude.

    :param pressure: Numpy array of floats representing pressure, in
        dbar or psi
    :param latitude: Latitude (-90.0 to 90.0)
    :param depth_units: 'm' for meters, 'ft' for feet. Defaults to 'm'.
    :param pressure_units: 'dbar' for decibars, 'psi' for PSI. Defaults
        to 'dbar'.

    :return: A numpy array representing depth in meters or feet
    """
    pressure = pressure_in.copy()
    if pressure_units == "psi":
        pressure /= DBAR_TO_PSI

    depth = -gsw.z_from_p(pressure, latitude)

    if depth_units == "ft":
        depth *= 3.28084

    return depth
```

Internal Temperature

```
def convert_internal_seafet_temperature(temperature_counts: np.ndarray):
    """Converts the raw internal temperature counts to degrees Celcius

    :param temperature_counts: raw internal temperature counts
    :return: internal temperature in Celcius
    """
    slope = 175.72
    offset = -46.85
    int_16bit = 2**16
    temperature = temperature_counts / int_16bit * slope + offset

    return internal_temperature
```

Internal Humidity

```
def convert_seafet_relative_humidity(humidity_counts: np.ndarray, temperature:
np.ndarray):
    """Convert relative humidity counts to percent

    :param humidity_counts: raw relative humidity counts
    :param temperature: converted internal temperature in Celcius
    :return: temperature compensated relative humidity in percent
    """
    slope = 125
    offset = -6
    int_16bit = 2**16
    max_humidity = 119
    temperature_coefficient = -0.15
    temperature_25c = 25

    # Uncompensated relative humidity
    relative_humidity = slope * humidity_counts / int_16bit + offset

    for n, humidity in enumerate(relative_humidity):
        # Theoretically, uncompensated relative humidity can be up to 119%
        if 0 <= humidity < max_humidity:
            relative_humidity[n] = humidity + temperature_coefficient * (
                temperature_25c - temperature[n]
            )

    np.clip(relative_humidity, a_min=0, a_max=100)

    return relative_humidity
```

Practical Salinity

Salinity calculation from the existing CTD ION_functions:

```
def ctd_pracsal(c, t, p):
    """
    Description:

         OOI Level 2 Practical Salinity core data product, which is
         calculated using the Thermodynamic Equations of Seawater - 2010
         (TEOS-10) Version 3.0, with data from the conductivity, temperature
         and depth (CTD) family of instruments.

    Implemented by:

        2013-03-13: Christopher Wingard. Initial code.
        2013-05-10: Christopher Wingard. Minor edits to comments.
        2014-01-31: Russell Desiderio. Standardized comment format.
```

```
    Usage:

        SP = ctd_pracsal(c, t, p)

            where

        SP = practical salinity, PSS-78, (PRACSAL_L2) [unitless]
        c = sea water conductivity (CONDWAT_L1) [S m-1]
        t = sea water temperature (TEMPWAT_L1) [deg_C]
        p = sea water pressure (PRESWAT_L1) [dbar]
    References:

         OOI (2012). Data Product Specification for Salinity. Document
         Control Number 1341-00040. https://alfresco.oceanobservatories.org/
         (See: Company Home >> OOI >> Controlled >> 1000 System Level >>
         1341-00040_Data_Product_SPEC_PRACSAL_OOI.pdf)
    """

   # Convert L1 Conductivity from S/m to mS/cm
    C10 = c * 10.0

    # Calculate the Practical Salinity (PSS-78) [unitless]
    SP = gsw.sp_from_c(C10, t, p)
    return SP
```

Density

Density calculation from existing CTD ION_functions:

```
def ctd_density(SP, t, p, lat, lon):
    """
    Description:

        OOI Level 2 Density core data product, which is calculated
        using the Thermodynamic Equations of Seawater - 2010
        (TEOS-10) Version 3.0, with data from the conductivity,
        temperature and depth (CTD) family of instruments.

    Implemented by:

        2013-03-11: Christopher Mueller. Initial code.
        2013-03-13: Christopher Wingard. Added commenting and moved
        to ctd_functions
        2013-05-10: Christopher Wingard. Minor edits to comments.
        2014-01-31: Russell Desiderio. Standardized comment format.

    Usage:

        rho = ctd_density(SP, t, p, lat, lon)

            where
```

```
        rho = sea water density (DENSITY_L2) [kg m-3]
        SP = practical salinity PSS-78 (PRACSAL_L2) [unitless]
        t = sea water temperature (TEMPWAT_L1) [deg_C]
        p = sea water pressure (PRESWAT_L1) [dbar]
        lat = latitude where input data was collected [decimal
        degree]
        lon = longitude where input data was collected [decimal
        degree]

    References:

        OOI (2012). Data Product Specification for Density.
        Document Control Number 1341-00050.
        https://alfresco.oceanobservatories.org/ (See: Company Home
        >> OOI >> Controlled >> 1000 System Level >>
        1341-00050_Data_Product_SPEC_DENSITY_OOI.pdf)
    """
    # Calculate the density [kg m-3]
    rho = gsw.ctd_density(SP, t, p, lat, lon)
    return rho
```

Dissolved Oxygen

Extracted from Sea-Bird conversion repo: (https://github.com/Sea-BirdScientific/seabirdscientific/blob/main/src/seabirdscientific/conversion.py)

```
def convert_sbe63_oxygen(
    raw_oxygen_phase: np.ndarray,
    thermistor: np.ndarray,
    pressure: np.ndarray,
    salinity: np.ndarray,
    coefs: Oxygen63Coefficients,
    thermistor_coefs: Thermistor63Coefficients,
    thermistor_units: Literal["volts", "C"] = "volts",
):
    """Returns the data after converting it to ml/l.

    raw_oxygen_phase is expected to be in raw phase, raw_thermistor_temp
    in counts, pressure in dbar, and salinity in practical salinity (PSU)

    :param raw_oxygen_phase: SBE63 phase value, in microseconds
    :param thermistor_temp: SBE63 thermistor data to use are reference,
        in counts
    :param pressure: Converted pressure value from the attached CTD, in
        dbar
    :param salinity: Converted salinity value from the attached CTD, in
        practical salinity PSU
    :param coefs (Oxygen63Coefficients): calibration coefficients for
```

```
        the SBE63 sensor

    :return: converted Oxygen value, in ml/l
    """
    if thermistor_units == "volts":
        temperature = convert_sbe63_thermistor(thermistor, thermistor_coefs)
    elif thermistor_units == "C":
        temperature = thermistor
    else:
        raise ValueError

    oxygen_volts = raw_oxygen_phase / OXYGEN_PHASE_TO_VOLTS  # from the manual

    ksv = coefs.c0 + coefs.c1 * temperature + coefs.c2 * temperature**2

    # The following correction coefficients are all constants
    sol_b0 = -6.24523e-3
    sol_b1 = -7.37614e-3
    sol_b2 = -1.0341e-2
    sol_b3 = -8.17083e-3
    sol_c0 = -4.88682e-7

    ts = np.log((KELVIN_OFFSET_25C - temperature) / (KELVIN_OFFSET_0C +
temperature))
    s_corr_exp = (
        salinity * (sol_b0 + sol_b1 * ts + sol_b2 * ts**2 + sol_b3 * ts**3) +
sol_c0 * salinity**2
    )
    s_corr = e**s_corr_exp

    # temperature in Kelvin
    temperature_k = temperature + KELVIN_OFFSET_0C
    p_corr_exp = (coefs.e * pressure) / temperature_k
    p_corr = e**p_corr_exp

    # fmt: off
    ox_val = (
        (((coefs.a0 + coefs.a1 * temperature + coefs.a2 * oxygen_volts**2)
        / (coefs.b0 + coefs.b1 * oxygen_volts) - 1.0) / ksv) * s_corr * p_corr
    )
    # fmt: on

    return ox_val


def convert_sbe63_thermistor(
    instrument_output: np.ndarray,
    coefs: Thermistor63Coefficients,
):
    """Converts a SBE63 thermistor raw output array to temperature in
    ITS-90 deg C.

    :param instrument_output: raw values from the thermistor
    :param coefs: calibration coefficients for the thermistor in the
        SBE63 sensor
```

```
    :return: converted thermistor temperature values in ITS-90 deg C
    """
    log_raw = np.log((100000 * instrument_output) / (3.3 - instrument_output))
    temperature = (
        1 / (coefs.ta0 + coefs.ta1 * log_raw + coefs.ta2 * log_raw**2 +
coefs.ta3 * log_raw**3)
        - KELVIN_OFFSET_0C
    )
    return temperature

def convert_oxygen_to_umol_per_kg(ox_values: np.ndarray, potential_density:
np.ndarray):
    """Converts given oxygen values to milligrams/kg.

    Note: Sigma-Theta is expected to be calculated via gsw_sigma0,
    meaning is it technically potential density anomaly. Calculating
    using gsw_rho(SA, CT, p_ref = 0) results in actual potential
    density, but this function already does the converison, so values
    will need to have 1000 subtracted from them before being passed into
    this function. The function is done this way to stay matching to
    Application Note 64, but the results of either method are identical.

    :param ox_values: oxygen values, already converted to ml/L
    :param potential_density: potential density (sigma-theta) values.
        Expected to be the same length as ox_values

    :return: oxygen values converted to milligrams/Liter
    """

    oxygen_umolkg = (ox_values * OXYGEN_MLPERL_TO_UMOLPERKG) /
    (potential_density + 1000)
    return oxygen_umolkg
```

pH Total

The more complex external pH extracted from Sea-Bird Application Note 99 and converted by Chris Wingard (OSU): (https://bitbucket.org/ooicgsn/cgsn-processing/src/master/cgsn_processing/process/proc_cphox.py)

```
def ph_total(vrs_ext, degc, psu, dbar, k0, k2, f):
    """
    Calculate the total pH from the SeapHOx sensor. The total pH is
    calculated from the external voltage (vrs_ext), temperature (degC),
    salinity (psu), pressure (dbar), and the calibration coefficients (k0,
    k2, f). Source is Sea-Bird Scientific Application Note 99, "Calculating
    pH from ISFET pH Sensors".

        :param vrs_ext: external voltage from the FET sensor
        :param degc: temperature in degrees Celsius
        :param psu: salinity in practical salinity units
        :param dbar: pressure in decibars
```

```
        :param k0: calibration coefficient from vendor documentation
        :param k2: calibration coefficient from vendor documentation
        :param f: calibration coefficients (f0, f1, f2, f3, f4, f5) from the
           vendor documentation (as an array)
        :return: pH total
        """
    fp = f[0] * dbar + f[1] * dbar**2 + f[2] * dbar**3 + f[3] * dbar**4 + f[4]
    * dbar**5 + f[5] * dbar**6

    bar = dbar * 0.10  # convert pressure from dbar to bar

    # Nernstian response of the pH electrode (slope of the response)
    r = 8.3144621      # J/(mol K) universal gas constant
    t = degc + 273.15  # temperature in Kelvin
    f = 9.6485365e4    # C/mol Faraday constant
    snerst = r * t * np.log(10) / f

    # total chloride in seawater
    cl_total = (0.99889 / 35.453) * (psu / 1.80655) * (1000 / (1000 - 1.005 *
    psu))

    # partial Molal volume of HCl (calculated as Millero 1983)
    vhcl = 17.85 + 0.1044 * degc - 0.0001316 * degc**2

    # Sample ionic strength (calculated as Dickson et al. 2007)
    i = (19.924 * psu) / (1000 - 1.005 * psu)

    # Debye-Huckel constant for activity of HCl (calculated as Khoo et al.
    # 1977)
    adh = 0.0000034286 * degc**2 + 0.00067503 * degc + 0.49172143

    # log of the activity coefficient of HCl as a function of temperature
    # (calculated as Khoo et al. 1977)
    loghclt = ((-adh * np.sqrt(i)) / (1 + 1.394 * np.sqrt(i))) + (0.08885 -
    0.000111 * degc) * i

    # log10 of the activity coefficient of HCl as a function of temperature #
    and pressure (calculated as Johnson et al. 2017)
    loghcltp = loghclt + (((vhcl * bar) / (np.log(10) * r * t * 10)) / 2)

    # total sulfate in seawater (calculated as Dickson et al. 2007)
    so4_total = (0.1400 / 96.062) * (psu / 1.80655)

    # acid disassociation constant of HSO4- (calculated as Dickson et al.
    # 2007)
    ks = (1 - 0.001005 * psu) * np.exp((-4276.1 / t) + 141.328 - 23.093 *
    np.log(t) + ((-13856 / t) + 324.57 - 47.986 * np.log(t)) * np.sqrt(i) +
    ((35474 / t) - 771.54 + 114.723 * np.log(t)) * i - (2698 / t) * i**1.5 +
    (1776 / t) * i**2)

    # partial Molal volume of HSO4- (calculated as Millero 1983)
    v_hso4 = -18.03 + 0.0466 * degc + 0.000316 * degc**2

    # compressibility of sulfate (calculated as Millero 1983)
    kbar_s = (-4.53 + 0.09 * degc) / 1000
```

```
    # acid dissociation constant of HSO4- as function of salinity,
    # temperature, and pressure (calculated as Millero 1982)
    kstp = ks * np.exp((-v_hso4 * bar + 0.5 * kbar_s * bar**2) / (r * t * 10))

    # calculate the pH total, adjusted for pressure, temperature and
    # salinity
    p_h = (((vrs_ext - k0 - k2 * degc - fp) / snerst) + np.log10(cl_total) + 2
    * loghcltp - np.log10(1 + (so4_total / kstp)) - np.log10((1000 - 1.005 *
    psu) / 1000))

    return p_h
```

Refer to Application Note 99 (https://www.seabird.com/cms-portals/seabird_com/cms/documents/Application-Note-Calculating-SeaFET-pH.pdf) for example values to test the ph_total portion of the algorithm.

IV. Port Agent & Driver

Port Agent

A new port agent instance shall be created on the Portland computer systems for each new PHSEN-H. Existing port agent code is sufficient. New configurations must be added to the supervisor configuration files in order to instantiate new port agent instances. For each each new PHSEN deployed, this will require:

- Reference designators
- IP addresses
- Data port numbers
- Control port numbers
- Driver path

Driver

Reference Designators

CE02SHBP-LJ01D-10-PHSENH110 and CE04OSPS-PC01B-4C-PHSENH109

Instrument Data Format & Particle Plan

OutputFormat=0 is raw data in decimal format.

Order in which the parameters show in the output: FrameSync, timestamp, data error flag, temperature, Vrs_ext, pH temperature, Vk, Ib, Ik, pressure, pressure temperature, conductivity, oxygen phase, oxygen temperature, internal relative humidity, internal temperature

Example output (from manual):

```
DSPHOX00113,2020-08-12T11:48:23, 0000, 474165, 5136915, 5085728,
8378529, 8383169, 525146, 1205, 5759.352, 19.285, 1.013468, 21472,
19648
```

Sample laboratory testing data:

```
DSPHOX02106,2025-01-29T22:52:00, 0000,  534641, 4639800, 5161011,
8379677, 8384971,  524650,  2299, 5135.465, 19.198, 1.104991, 19740,
3772
```

Parameters to configure prior to deployment:

OutputExecutedTag=Y
TxRealTime=Y
OutputFormat=0
SampleInterval =120s (6-21600 s)
Setoxunits = 0 (0=ml/L, 1=mg/L) #likely does not matter for outputformat=0, but no harm in setting it
PumpTime=? (0-550)
Ctdpower=1 (power CTD from external source)

Sampling should be no less often than the SAMI that it replaced (15 mins), but ideally as frequently as possible to avoid any detrimental effects to the data or instrument. Because the SBE 37-SMP-ODO pump operates before each burst of data collection, the manufacturer recommends that the shortest data collection interval is every two minutes (120 sec) for the SeapHOx.

Detailed Data String Formats

String formats are as follows:

Header String

Engineering String

Data String

Data String Parsing

The following regular expression can be used to capture a full data “block”, assigning the six different data strings to a named match group:

Header Data

Engineering Data

Science Data

V. Raw Data Logging

All raw data and commands sent to the PHSEN will be captured in logs saved on the UW Salacia data server and can be used for data playback as needed. Loggers will record daily files for each active TCP port connection, including commands sent to the instrument and data returned by the instrument.

VI. Outstanding Questions & Concerns

Questions

-

Answered Questions

-

Appendix I. - Other Resources

Redmine Tickets

- Driver Development: #15983
- Telemetered support: #15948

Google Drive Folders

- UW PHSEN-H Dev Docs

Example code

- CGSN cphox code: https://bitbucket.org/ooicgsn/cgsn-processing/src/master/cgsn_processing/process/proc_cphox.py

Manuals

- Deep SeapHOx V2™ pH, C, T, P, DO (seabird.com)
    - Document No.: Deep SeapHOxV2
    - Release Date: 2023-09-07
    - Version: C
    - Software: UCI 2.04
- SeaBird Application Note 99 Calculating pH from ISFET pH Sensors
    - https://www.seabird.com/cms-portals/seabird_com/cms/documents/Application-Note-Calculating-SeaFET-pH.pdf