

SeeStar III Sensor Module

by

Matthew Ducasse

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

June 14, 2017

Table of Contents

<i>Section</i>	<i>Pages</i>
List of Tables and Figures	2
Abstract	3
<i>Chapters</i>	
I. Introduction	1
II. Customer Needs, Requirements and Specifications	2
III. Functional Decomposition (Level 0 and Level 1)	4
IV. Project Planning (Gantt Chart and Cost Estimates)	7
V. Project Design	9
VI. Project Testing	14
VII. Works Cited	16
<i>Appendices</i>	
A. ABET Senior Project Analysis	17
B. Integrating New Sensor Types into the SeeStar III Sensor Module Software	20

List of Tables

<i>Table</i>	<i>Pages</i>
I. Requirements and Specifications	2
II. Project Deliverables	3
III. Module Level-0 Functionality Table	4
IV. DC-DC Converter Level-1 Functionality Table	5
V. Microcontroller Level-1 Functionality Table	6
VI. Instrument Control Level-1 Functionality Table	6
VII. Estimated Project Costs Breakdown	8
VIII. List of Config File Keywords	11
IX. List of Serial Commands	13
X. Specification Test Results	15

List of Figures

<i>Figure</i>	<i>Pages</i>
I. Level-0 Block Diagram	4
II. Level-1 Block Diagram	5
III. Project Timeline Gantt Chart	7
IV. PERT Cost Estimation Formula	8
V. Config File Example	11
VI. Software Flowchart	13

Abstract

SeeStar is an autonomous, underwater camera system for observing aquatic environments. The Monterey Bay Aquarium Research Institute (MBARI) developed SeeStar to provide a low cost solution for marine researchers wishing to observe underwater ecosystems over long durations. The system consists of completely open-source designs featuring only common, commercially available components. This means users may purchase the components and build the system themselves. While MBARI originally intended for the system to simply capture photo and video data, many customers have requested additional sensors to gather contextual data to complement the captured images. The latest generation, SeeStar III, features a sensor module, enabling users to easily integrate sensors into the system.

This project phase encompasses sensor module firmware design and testing. The primary design challenge is creating a general interface so users may integrate additional sensors without modifying the system's firmware for each new sensor. This preserves the system's ease-of-use. The secondary design challenge is maximizing power efficiency such that long term untethered deployments are possible.

Chapter 1: Introduction

The climate challenges of the 21st century have led to an unprecedented need for marine-biological research. Observing the behavior of marine life can hold valuable insight for the state of the planet and our well-being. However, many marine researchers had difficulty collecting such oceanographic data because most underwater imaging equipment is expensive and designed for niche applications. They needed an underwater imaging system that was inexpensive, easy to assemble, and versatile.

The Monterey Bay Aquarium Research Institute (MBARI) designed and released the SeeStar underwater imaging system in 2014. MBARI designed SeeStar to meet the needs of many researchers by featuring a simple, low-cost design that is easy to assemble, deploy, and maintain. The system features a modular design, consisting of a waterproof camera, LED light, and a battery. The system design is completely open-source, and can be assembled by the user with common, commercially available components. The camera module also contains a microcontroller that controls the system in untethered deployments, and activates the camera and LED on timed intervals.

The latest generation of the SeeStar system, the SeeStar III, features a sensor module to allow for contextual data collection upon image captures. Many users of the previous generations requested the ability to collect data along with their images, such as temperature, salinity, and pH. This data can greatly increase the insight provided with each photo by adding contextual information. For instance, a researcher could correlate the number of fish visible in a series of photos with the temperatures measured at the times each photo was captured.

To maximize the sensors module's utility, it features three general purpose sensor ports where users can plug in virtually any aquatic sensor that has a serial output. This allows different users to tailor the system for their research, or to reuse the system in different experiments. The system can be easily configured for each new sensor by entering the sensor's parameters into the system's console interface. The SeeStar also features built-in temperature and pressure sensors.

Chapter 2: Customer Needs, Requirements and Specifications

Customer Needs Assessment

Many marine researchers need an inexpensive, autonomous underwater camera system to monitor underwater environments. MBARI met this need with the low-cost SeeStar solution. Many customers used the SeeStar system to study underwater biological activity, such as feeding patterns and population numbers. Customer feedback indicated that many studies could benefit from additional oceanic data (including temperature, pressure, pH and salinity) to contextualize their captured images. The SeeStar III system features a sensor module, allowing users to integrate sensors with the system and take measurements during deployments. The sensor module features the same modular, low-cost design as the rest of the system, so low-budget customers may still afford it.

Requirements and Specifications

The customer requests collected after SeeStar II's release drove the design of the new feature set in the SeeStar III system. One of the most common requests was the ability to collect contextual data when capturing photo or video. The new system features a sensor module to provide contextual data. To maintain simplicity, accessibility and utility to all users, the sensor module is designed for compatibility with a wide range of sensors (Table I, requirement 1). This way, users can use the sensor module in a wide variety of experiments by swapping out the sensors. In addition, the user can configure the capture interval, to optimize data resolution and power consumption (Table I, requirements 5, 6). The sensor module also features a built in temperature sensor and pressure sensor (Table I, requirement 2). Not only are temperature and pressure useful in most oceanographic studies, but these variables can affect the performance of the system's electronics. Temperature and pressure measurements can be used by the system for self-calibration. Table I lists all of the project's marketing requirements and engineering specifications.

TABLE I
SEESTAR III SENSOR MODULE REQUIREMENTS AND SPECIFICATIONS

Marketing Requirements	Engineering Specifications	Justification
1, 5	General sensor ports must read from any serially interfacing sensor at baud rates between 1K and 1M baud.	The SeeStar system may find use in several applications. Users select different sensors to conduct different experiments.
1, 5	General sensor ports must power any serially interfacing sensor operating between 5V and 12V, and under 2A.	The SeeStar system may find use in several applications. Users select different sensors to conduct different experiments.
1	Must accept up to three peripheral sensors simultaneously.	Users may need to collect multiple data plots in one experiment.
2	Must measure temperatures between -5°C and 35°C. Temperature measurement error must not exceed $\pm 0.005^\circ\text{C}$.	Many applications may require temperature measurements. Temperature values also effect the module's operation.
2	Must measure pressures up to 682 PSI. Pressure measurement error must not exceed	Oceanic pressure correlates directly to depth. Many applications may require depth

	±0.05 PSI.	information.
3	Must withstand any depth not exceeding 1500 feet.	Some applications may require deep water observation.
4	Must accept any supply voltage between 14V and 18V.	The SeeStar system uses an 18V battery. The sensor module must use the system's supply.
5	Capture rate must be user-adjustable between 1 capture/day and 4 captures/minute.	Different applications require different data resolutions. Capture rate also effects power consumption, so users may optimize it for their needs.
6	Power consumption must not exceed 0.5W with a capture rate of 1 capture/hour or less.	A battery may supply the power. A longer battery lifetime makes maintenance easier.
6	Power consumption must not exceed 5W with a capture rate of 12 capture/hour or less.	A power cord may supply the power. Still, the system should operate efficiently in higher power settings.
7	The cost of parts must not exceed \$500.	The sensor module addition must not make the system unaffordable.
7, 8	Must only contain common, commercially available microcontroller(s) and circuit components.	This makes it much easier for customers to acquire and replace system components.
9	Chassis size must not exceed 5" x 4" x 2.5".	A smaller chassis size enables easy system integration.

Marketing Requirements

1. Three general-purpose sensor ports.
2. Built-in temperature and pressure measurement
3. 1500 feet maximum depth.
4. Easy system integration.
5. Configurable.
6. Low power.
7. Low cost.
8. Easy assembly and maintenance.
9. Small form factor.

TABLE II
SEESTAR III SENSOR MODULE PROJECT DELIVERABLES

Delivery Date	Deliverable Description
2/16/17	Design Review
3/8/17	EE 461 demo
3/11/17	EE 461 report
5/3/17	EE 462 demo
4/29/17	ABET Sr. Project Analysis
5/20/17	Sr. Project Expo Poster
5/29/17	EE 462 Report

Chapter 3: Functional Decomposition (Level 0 and Level 1)

Level 0 Block Diagram

The Sensor Module is an interface between the SeeStar system and any aquatic sensor the user wishes to use. The module is powered by 14-18V DC. The module internally regulates the supply voltage down to 5V and 12V to power internal components and peripheral sensors. The sensor module records sensor measurements each time a “capture” signal is received from the camera module and stores the measured data internally. Figure I shows a level 0 block diagram of the sensor module.

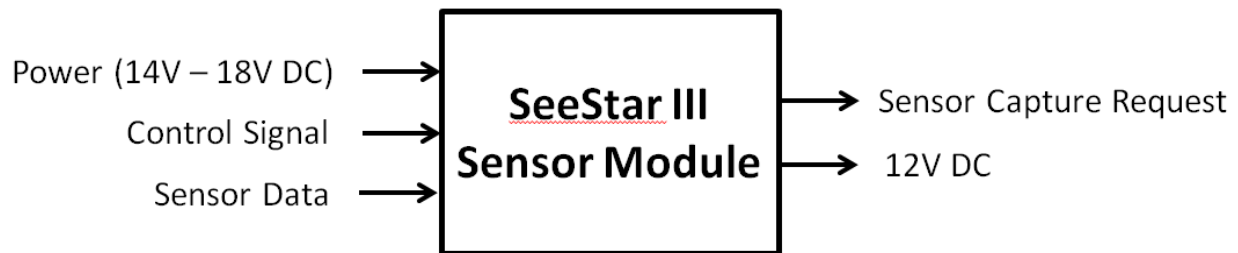


FIGURE I
SEESTAR III SENSOR MODULE LEVEL-0 BLOCK DIAGRAM

TABLE III
SEESTAR III SENSOR MODULE LEVEL-0 FUNCTIONALITY TABLE

<i>MODULE</i>	Sensor Module for SeeStar III Underwater Camera
<i>INPUTS</i>	<ul style="list-style-type: none"> • Power: 14-18V DC • Camera module control signal • Sensor input as streaming data or polled data
<i>OUTPUTS</i>	<ul style="list-style-type: none"> • Control signal to sensors to initiate data streaming or polling • 12V DC to power peripheral oceanographic sensors
<i>FUNCTIONALITY</i>	Integrate 3 rd -party sensors with SeeStar camera for contextual imaging data. Convert the SeeStar’s 14-18V DC battery module to 5-12V DC for 3 rd -party sensor power supply. Store sensor data on MicroSD card.

Level 1 Block Diagram

The SeeStar III Sensor Module consists of three main components: a voltage regulator, a microprocessor, and an instrument control interface. Figure II shows level 1 diagram highlighting the sensor module's subsystems.

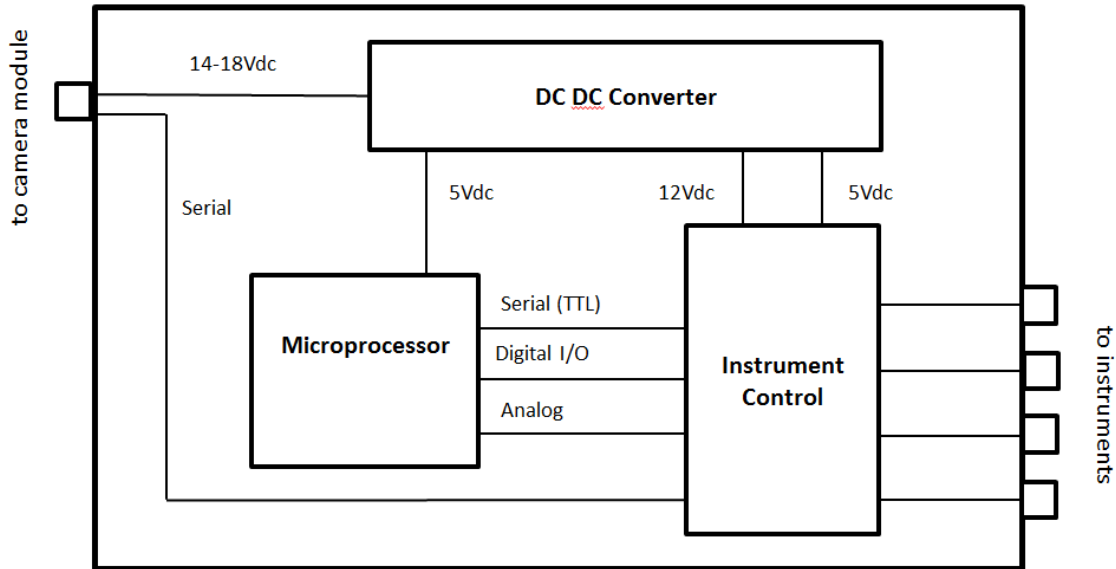


FIGURE II
SEESTAR III SENSOR MODULE LEVEL-1 BLOCK DIAGRAM

The DC-DC Converter regulates the system's supply voltage, outputting 5V and 12V. The 5V supplies the module's microprocessor and components in the instrument control block. The 12V output powers peripheral sensors. Table IV outlines the DC-DC Converter's features.

TABLE IV
DC-DC CONVERTER LEVEL-1 FUNCTIONALITY TABLE

<i>MODULE</i>	DC-DC Converter
<i>INPUTS</i>	<ul style="list-style-type: none"> • Power: 14-18V DC
<i>OUTPUTS</i>	<ul style="list-style-type: none"> • 5V DC • 12 VDC
<i>FUNCTIONALITY</i>	Convert 14-18V battery power supply to 12V and 5V levels. Supplies lower voltages to other system components.

The module's internal microprocessor controls measurement sequences. When the module receives a "capture" signal, the microprocessor collects raw sensor data from the instrument control block, decodes the data, and saves it to memory. Table V outlines the microprocessor block's features.

TABLE V
MICROCONTROLLER LEVEL-1 FUNCTIONALITY TABLE

<i>MODULE</i>	Microcontroller
<i>INPUTS</i>	<ul style="list-style-type: none"> • Power 5V DC
<i>OUTPUTS</i>	<ul style="list-style-type: none"> • Serial (TTL) • Digital I/O • Analog Signals
<i>FUNCTIONALITY</i>	Control system, calibrate sensors and record sensor data.

The instrument control block is an interface between the microcontroller and the peripheral sensors. It provides ports for three digital output sensors, one analog output sensor, and two built-in sensors (temperature and pressure). It also controls power supplied to each sensor. Table VI outlines the instrument control block's features.

TABLE VI
INSTRUMENT CONTROL LEVEL-1 FUNCTIONALITY TABLE

<i>MODULE</i>	Instrument Control
<i>INPUTS</i>	<ul style="list-style-type: none"> • Power 12V DC • Power 5V DC • Sensor Data
<i>OUTPUTS</i>	<ul style="list-style-type: none"> • 12V DC • Sensor Capture Signals
<i>FUNCTIONALITY</i>	Supply power to sensors and route sensor data to microcontroller.

One preparation phase and three design-test-build iterations constitute the project’s timeline. The preparation phase includes research, high-level design, cost estimation and ABET analysis. The first design phase focuses on investigating the feasibility of each major firmware function (variable sensor interfacing, minimal power usage, etc). The second design phase involves designing the full system, focusing on reliable operation and interfacing between components. The third design phase involves modifying firmware to optimize speed, power usage and reliability.

Cost Estimate

The project’s firmware development phase is relatively inexpensive. Since the hardware was completed in the previous project phase, the only hardware expenses come from additional sensors purchased to test the peripheral sensor interfacing. The primary expense in this phase is engineering. Costs estimates are based on the PERT formula shown in Figure IV.

$$C_{est} = \frac{C_{opt} + 4C_{real} + C_{pes}}{6}$$

C_{est} = cost estimate

C_{opt} = optimistic cost

C_{real} = realistic cost

C_{pes} = pessimistic cost

FIGURE IV
PERT COST ESTIMATION FORMULA

TABLE VII
ESTIMATED PROJECT COSTS BREAKDOWN

<i>Item Name</i>	<i>Cost/Unit (Optimistic)</i>	<i>Cost/Unit (Realistic)</i>	<i>Cost/Unit (Pessimistic)</i>	<i>Cost / Unit (Estimated)</i>	<i>Unit Qty</i>	<i>Subtotal</i>
Parts						
pH Sensor	\$20	\$30	\$45	\$31	1	\$31
Temp. Sensor	\$5	\$12	\$25	\$13	1	\$13
Light sensor	\$3	\$6	\$12	\$7	1	\$7
Engineering						
Firmware Design / Testing	\$20	\$35	\$55	\$35/hr	155	\$5,425
TOTAL						\$5,476

Chapter 5: Design

Hardware Selection

The hardware system included within the scope of this senior project can be divided into three main functional blocks, the Camera Module, the Sensor Module, and the Sensor Modules built-in sensors. While the hardware component selection was completed in a previous senior project, it is relevant to this project.

The Camera Module is built around a Raspberry Pi 3. The Raspberry Pi includes many features needed in the Camera Module, including SD card compatibility, serial communication pins, and a powerful processor. However, the Raspberry Pi does not feature a low power mode needed in a battery operated application like See Star, so a Sleepy Pi accessory has been integrated with the Raspberry Pi to enable full sleep between measurements. In addition, the Pi's serial pins are connected to an external TTL to RS232 converter to enable communication with the Sensor Module's Arduino.

The Sensor Module uses an Arduino Mega 2560 microcontroller. The Mega features a wide array of external pins, including SPI, PWM, serial UART, and several DIO pins. This allows for significant flexibility when developing the Sensor Module to operate multiple peripheral sensors simultaneously. In addition, a Micro SD Shield has been integrated with the Arduino Mega to enable interfacing with an SD card. The primary drawback of the Mega 2560 is its high power usage. As this project is an early rendition of the See Star III system, the Sensor Module software will eventually need to be ported to a lower power microcontroller before the system can be feasibly deployed for long durations on a single battery.

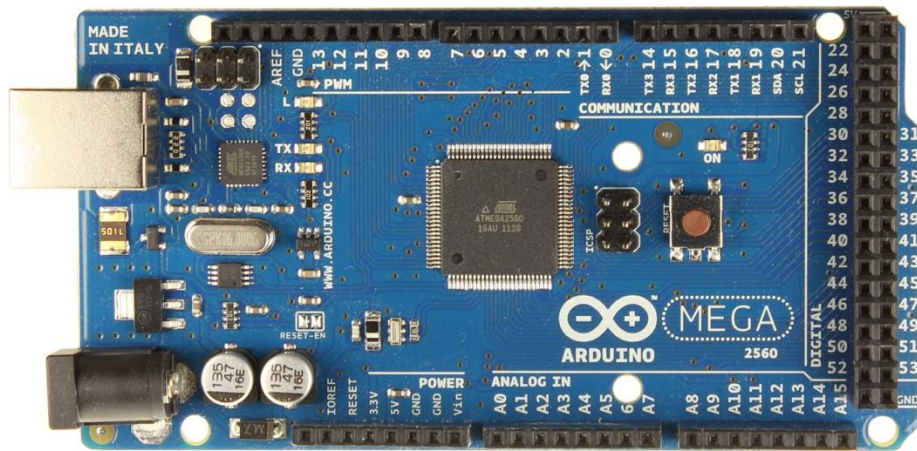


Figure X. Arduino Mega 2560 microcontroller.

The Sensor Module also needs to interface with up to 4 peripheral serial sensors. The Sensor Module uses MAX3232 TTL to RS232 converters to translate the voltage levels of the peripheral sensor's serial stream down to the 0 – 3.3V levels used by the Arduino's IO pins. Since the MAX3232 board can convert two serial interfaces, 2 boards are needed to accommodate all 4 serial channels. To reduce total power draw due to sensor operation, the sensors must be powered down completely between measurements. This is achieved with the iTead 4-channel relay board. Each relay completes a connection with one of the peripheral serial sensors, and is actuated by one of the DIO pins on the Arduino Mega.

Finally, the Sensor Module features two built-in sensors to measure temperature and pressure. The temperature sensor is a 5V Omega temperature sensor, and it is measured via a Sparkfun MAX31855k SPI thermocouple breakout board. The pressure sensor is an 12V Omega pressure sensor, and interfaces with the Arduino with an Adafruit ADS 1115 4-channel I2C ADC.

Software Design

The sensor module is built around an Arduino Mega 2560. This board features several UART and DIO ports, ideal for interfacing with several sensors. An SD card “shield” is integrated with the Mega to allow interfacing with a Micro SD card. In addition, the Arduino interfaces with a Sparkfun MAX31855k SPI thermocouple to gather data from a temperature sensor, and an Adafruit ADS1115 ADC to read a pressure sensor. Finally, the serial UART ports on the Mega provide multiple channels to connect different serially controlled sensors.

The sensor module features 10 channels, which are software structures that map to a physical sensor. To inform the Arduino what kinds of sensors are connected to it, it's SD card must be loaded with “config files” before startup. For each channel to be used, the user will write one config file describing the type of sensor, and the physical pins or port that it will utilize. Upon startup, the sensor module will read these config files, and initialize each channel with the information in its file, and store this information in a data structure for use throughout the module's operation.

The information listed in a config file is dependent on the type of sensor it is describing. For instance, a serial sensor's config file should indicate which of the Arduino's serial UARTS it will use for communication, while an SPI should indicate which of the Arduino's DIO pins it will use for VCC, GND, and CS. In other words, the config file will list values that will vary between sensors of the same type, such as what pins it uses. On startup, the Sensor Module's software will determine what type of sensor a config file describes by reading its first line. A config file always starts with the keyword “type”, followed by a word indicating the type of sensor (e.g. “ser”, “spi”, “i2c”, etc.). After the sensor type is determined, a function is called that searches the config file for information specific to that sensor type. It does this by searching for pre-defined keywords, and reads the values following each keyword. A list of keywords specific

to each build-in sensor type is shown in Table VIII below. An example config file is shown in Figure V.

Table VIII. List of keywords used by config files for each sensor.

Serial Sensor Keywords	I2C Sensor Keywords	SPI Sensor Keywords
powerpin	adcchannel	cspin
hwserial		vccpin
swserialrx		gndpin
swserialtx		
delimiter		
delimiter2		

```

1 type spi
2 cspin 43
3 vccpin 40
4 gndpin 41
5

```

Figure V. An example of a config file for an SPI sensor. Note by the name that this file would be used to configure channel 2.

The camera module will poll the sensor module regularly to gather sensor data or other information. To facilitate this, the system uses a serial data protocol to structure requests from the camera module and replies from the sensor module. The format of camera module requests and the set of valid opcodes is outlined in Figure VI. The “destination ID” field indicates which module the packet is intended for. The camera module’s ID is always 0, and connected sensor modules are given a unique ID of 1-9. In systems with multiple sensor modules, requests are broadcast on all channels connected to sensor modules, and only acknowledged by the sensor module with the correct ID.

Table IX. Camera Module request format and list of opcodes.

\$ <destination ID> <opcode> <channel>		(i.e.) \$1rd0	
PA	Power All Channels	RF	Read Configuration
PC	Power Channel	RS	Read Status
RD	Read Channel	WL	Write Calibration
RB	Read Block	WF	Write Configuration
RL	Read Calibration	RE	Reboot System

Since the sensor module can interface with serial, I2C, and SPI sensors, three methods of measurement were implemented in the modules firmware. Sensor measurements are made with Arduino’s “Serial” API, I2C measurements are made with Adafruit’s “ADS” API, and SPI measurements are made with Sparkfun’s “Probe” API. “Serial” only provides functionality for reading single bytes, rather than discrete measurements. Therefore, I implemented a function to read bytes from the sensor once a “delimiter” character is detected, and continue reading until a second “delimiter” is read. This delimiter character is a specific character that the sensor uses to indicate the end of a data packet, and is referenced by the user in that sensors config file.

The sensor module will remain in low power mode until it receives a serial transmission from the camera module. To reduce errors, all serial transmissions from the camera module are stored in a special array in the sensor module called the “Serial Buffer”. This buffer won’t accept any characters until a ‘\$’ is detected, then it reads the 4 subsequent characters. This ensures that requests won’t be operated upon unless they assume the proper format. If the buffer doesn’t reach 4 characters, a timeout will occur after 500 ms, and the serial input will be assumed to be erroneous. Then the buffer will be cleared and the sensor module will go into low power mode. A flowchart outlining the systems high-level operation is shown in figure VII.

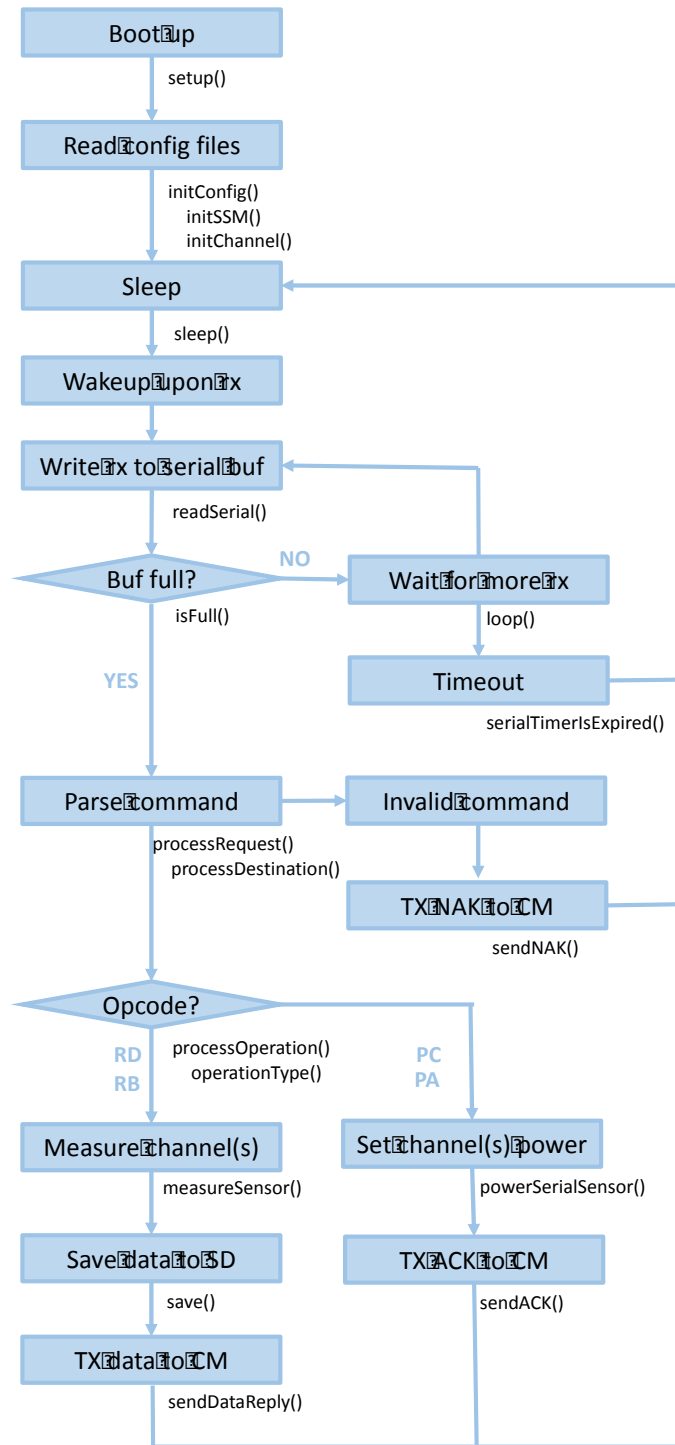


Figure VI. Flowchart of Sensor Module software operation. Note that the function calls associated with each state are listed to the lower right.

Chapter 6: Testing

I conducted a test while integrating each piece of hardware to ensure proper functionality and reliability. First, I tested the measurement functionality of the Sensor Module. I achieved this by connecting an RBR temperature sensor to the Sensor Module, and wrote a config file with the sensor's information. Next, I powered on the Sensor Module's Arduino and entered the command `$1RD0` in the Arduino IDE's serial input field, to indicate that I wanted to read data from the Sensor Module's channel 0 sensor. After successfully reading the room's temperature, I integrated the Omega temperature and pressure sensors to prove that the system can handle multiple sensors. I entered the commands `$1RD0`, `$1RD1`, and `$1RD2` to read data from each sensor individually. Next, I entered `$1RB0` to read a data block (data from every sensor). After the test, I inspected the contents of the system's SD card and found that all of the measured data was saved successfully.

The next step in system integration was connecting the Raspberry Pi such that it could send commands to the Arduino serially. First, I connected the serial IO pins on the Pi to the Serial 1 pins on the Arduino (with TTL to RS232 converters on both ends). I started by sending a simple ASCII string to verify a good connection. Next, I changed the Pi's program to print the response to each command I sent, and began to send commands in the proper format (e.g. `$1RD0`). Once I could reliably send commands and read measured data from the Pi's terminal, I began to work on a more robust serial driver for the Pi's communication with the Arduino.

The third test involved integrating the relays that toggled power to the serial sensors. First, I connected one of the relays to a DIO pin on the Arduino, and toggled power the pin to verify that the relay would open and close. Next, I wrote a function to the Sensor Module's software that could actuate a relay for a given channel (using the "powerpin" number indicated in the channels config file). I tested the function by entering `$1PC0` to toggle channel 0's power on/off. After I set up additional relays on other channels, I entered `$1PA1` (power all to state 1) and observed all of the relays' LEDs switching on.

Finally, I tested the full system by configuring all three sensors to the Sensor Module and connecting the serial interface to the Pi. I wrapped my serial driver with a simple text prompt user interface. This allowed any users unfamiliar with the serial protocol command format to interact with the system. I used the test program to test all of the currently supported commands (read data, read block, power channel, power all) and found that each worked reliably. The read data and read block commands would return measured data from the indicated sensors, and the power commands would successfully switch the relays.

Table X. Results of testing the predefined engineering specifications.

Result	Engineering Specifications	Explanation
Not tested	General sensor ports must read from any serially interfacing sensor at baud rates between 1K and 1M baud.	Only tested with a baud rate of 9600 (successfully).
Not tested	General sensor ports must power any serially interfacing sensor operating between 5V and 12V, and under 2A.	
Passed	Must accept up to three peripheral sensors simultaneously.	Tested with 2 serial sensors, 1 I2C sensor and 1 SPI sensor
Failed	Must measure temperatures between -5°C and 35°C. Temperature measurement error must not exceed $\pm 0.005^\circ\text{C}$.	Errors were due to sensor variations
Not tested	Must measure pressures up to 682 PSI. Pressure measurement error must not exceed ± 0.05 PSI.	
Not tested	Must withstand any depth not exceeding 1500 feet.	Not within project scope
Passed	Must accept any supply voltage between 14V and 18V.	
Not tested	Capture rate must be user-adjustable between 1 capture/day and 4 captures/minute.	Not within project scope
Failed	Power consumption must not exceed 0.5W with a capture rate of 1 capture/hour or less.	Power consumption exceeded 0.5W due to high power draw of Arduino Mega
Passed	Power consumption must not exceed 5W with a capture rate of 12 capture/hour or less.	
Passed	The cost of parts must not exceed \$500.	Excluding optional sensors, only includes sensor module hardware
Passed	Must only contain common, commercially available microcontroller(s) and circuit components.	
Not tested	Chassis size must not exceed 5" x 4" x 2.5".	Not within project scope

Chapter 7: Works Cited

1. Springer, Electronic System Level Design: An Open-source Approach. Dordrecht ; New York: 2011. Available: <http://www.springer.com/us/book/9781402099397>
2. Miao Yang and A. Sowmya , An Underwater Color Image Quality Evaluation Metric [online] Nov 12th 2015. IEEE Transactions on Image Processing, Volume 24, Issue 12, pages 6062 - 6071, Issued Dec 2015.
3. Atmel ATmega640/V-1280/V- 1281/V-2560/V- 2561/V Datasheet Datasheet by manufacturer Atmel. Updated Feb 2014. Available: http://www.atmel.com/images/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf
4. Christopher A. Scholin, Eugene Massion, David K. Wright, Dannelle E. Cline, Ed Mellinger and Mark Brown, “Aquatic autosampler device”, Patent No. 6,187,530. Filed Jul 15, 1999. Accepted Feb 13, 2001.
5. Lloyd Breslau. Underwater Camera, Patent No. 4,153,357. Filed Sept 22, 1977. Accepted May 8, 1979.
6. François Cazenave, Chad Key, Michael Risi, Steven H.D. Haddock, SeeStar: a low-cost, modular and open-source camera system for subsea observations. IEEE Xplore conference, Printed Sept 14, 2014. Available: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7003077>
7. Jean-Michel Leconte, “Logger2 Command Reference” Jan 5, 2016
8. Kim Fulton-Bennett, New SeeStar camera system allows researchers to monitor the depths without sinking the budget, [online] January 21 st , 2015 Available: <http://www.mbari.org/new-seestar-camera-system-allows-researchers-to-monitor-the-depths-without-sinking-the-budget/>
9. Texas Instruments Inc, “MSP432P4xx Family Technical Reference Manual” Apr 2015 Available: <http://www.ti.com/lit/ug/slau356a/slau356a.pdf>
10. NOAA Facts: CTD - Conductivity, Temperature, and Depth instruments <http://oceanexplorer.noaa.gov/facts/ctd.html>
11. Atmel blog: Printed 2/2/15 [online] Available: <http://blog.atmel.com/2015/02/02/seestar-is-an-open-source-camera-system-for-underwater-exploration/>

Appendix A: Senior Project Analysis (ABET Analysis)

Project Name: SeeStar III Underwater Camera Sensor Module

Student's Name: Matt Ducasse

Student's Signature: _____

Advisor's Name: Bridget Benson

Advisor's Initials: _____

ABET Senior Project Analysis

1. Summary of Functional Requirements

The SeeStar Sensor Module addition gathers oceanic measurements while capturing photo or video data. The additional data contextualizes the image content and enables researchers to deploy the SeeStar system as a scientific instrument in many more scenarios. The sensor module interfaces with several varieties of sensors with minimal configuration per sensor. The system gathers image and measurement data automatically, and stores the timestamped data. The Sensor Module features an integrated temperature sensor, and pressure sensor, and four general sensor ports.

2. Primary Constraints

Cost is the primary design and manufacturing constraint. The SeeStar system provides a low-cost alternative to expensive underwater imaging equipment. The Sensor Module must use only inexpensive, commercially available components.

3. Economic

The SeeStar system has a significant economic impact on individuals and companies involved in marine research. The system's low cost provides opportunities for many researchers unwilling to purchase more expensive equipment. The system also impacts its component's manufacturers, and shipping companies.

MBARI provided the hardware components necessary to develop and assemble the SeeStar prototype. MBARI also spent several hundred hours developing the prototype.

4. If manufactured on a commercial basis:

The Sensor Module hardware development costs \$6,000, and firmware development costs \$5,476. Each module costs \$600 to produce [6]. The production cost consists of \$200 for manufacturing and \$400 for parts. If sold for \$800, MBARI would break even after selling 58 units. If the SeeStar system were in high demand and could be sold in greater quantities, the manufacturing price would fall to \$500. The Module could be sold for \$515 and would break

even after 765 units were sold. The manufacturing costs for the rest of the system are \$700, and parts are \$2200. The total system would sell for \$4000.

The system's operation cost is relatively low. The main costs arise from operator salary, as a user must configure the system and extract measurement data between deployments. The battery also must be charged between deployments. Replacement part costs constitute the maintenance costs.

5. Environmental

SeeStar's environmental impacts include component manufacturing. The system's components each contain plastics and/or metals that are environmentally damaging to produce. The system could also interrupt the underwater ecosystem if not retrieved after use. Pollution from shipping system parts is another indirect environmental impact.

The positive environmental effects are less direct but potentially more numerous. The system is used by marine researchers to gather environmental data. This data could provide insight into the state of the marine ecosystem and how to prevent further degradation [8].

6. Manufacturability

The Sensor Module's hardware consists entirely of common components available from multiple brands. Mechanical hardware components (PVC pipe, screws, etc) are available at most hardware stores. Electrical hardware components are available on most online electronics retailers. Users may access MBARI's SeeStar designs, purchase the components and build the system themselves. The system's software may inhibit manufacturability. The software associated with the cameras, microcontrollers, and battery management may vary between manufacturers. This could affect the interfacing between modules and may require software design changes.

7. Sustainability

The system deploys in saltwater environments regularly, so parts may degrade and need replacement. The system's modular design allows for easy maintenance. Users can remove parts and purchase and install replacements easily. In addition, software updates may become available after purchase. Users can download software updates for no cost. Users may even integrate their own software or hardware upgrades.

Energy is required to produce and ship the system's components. However, system development and assembly mostly involves only human labor, and has a minimal environmental impact.

8. Ethical

Customers use the SeeStar system to collect data in scientific experiments with ecological implications. It is imperative that the customers are informed SeeStar Sensor Module's degree of accuracy, so they do not draw false conclusions from their data. This honesty to the customers embodies utilitarian ethics, as it acknowledges that correct scientific data can lead to knowledge that benefits a large number of people.

SeeStar also aims to comply with the IEEE code of ethics. Part 1 of the IEEE code of ethics states "to accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment". SeeStar is tested and assembled in a way that has minimal environmental impact. However, the systems contents may act as pollutants if not removed from the water after use. Users are notified to use the system responsibly and to not use it in situations that may harm the environment or marine organisms.

9. Health and Safety

The electrical connections between the system's components may threaten users and nearby organisms. Constant use in saltwater conditions can degrade protective casings, exposing conductors. The documentation must instruct users to regularly inspect the system and replace damaged components.

10. Social and Political

The production and use of the SeeStar system probably has little direct social and political impact. It requires few resources to manufacture, and it targets a relatively small customer base. It may have larger indirect consequences, since the system gathers oceanic environment data. This data may change how the community comprehends the underwater ecosystem, and affect how people interact with the environment.

11. Development

Planning this project involved studying the previous SeeStar generations, and familiarizing myself with the SeeStar III Sensor Module hardware. I also discussed firmware design goals with engineers from MBARI. I employed the Agile design process to develop the firmware, separating the project into three design/test/build phases. This made it easier to uncover unanticipated challenges and react sooner.

See Chapter 5: Works Cited for literature search.

Appendix B: Integrating New Sensor Types into See Star III Sensor Module Software

1. Preparation

1.1 Things to Know in Advance

Before attempting to modify the SSM software to interface with a new sensor type, it is important to know the following about the sensor:

- What type of interface it uses (Serial, SPI, I2C, etc.)
- What pins on the Arduino it could viably connect to.
- What software driver it uses.
- Whether it needs an “init” function to be called on system startup.
- How its measurement function(s) work.
- The datatype and format of its measurement function(s) output.
- Whether the sensor’s power is automatically or manually switched on/off.

1.2 Overview

Modifying the software to be compatible with a new sensor requires the following steps:

- Defining a new struct to hold parameters associated with the new sensor
- Defining a measurement function for gathering data from the sensor and converting it to string format
- Defining a configuration function that reads a config file for the new sensor and maps its contents to the sensor’s struct.

2. Creating a New Sensor Type Struct

2.1 Background

The Sensor Module uses a data structure (called SSM) to keep track of each of its channels. It contains information on what type of sensor each channel uses, whether the channel is enabled, what pins the channel’s sensor uses, etc.

From a code perspective, the SSM struct contains an array of Channel structs representing each of the Sensor Modules channels. Each Channel struct contains an SensorType enum indicating what kind of sensor that channel is interfacing with, and a Sensor union that can hold any sensor struct.

A sensor struct contains parameters unique to a specific type of sensor. For instance, the SerialSensor struct contains information like what pin controls the sensor’s power relay, which of the Arduino’s serial interfaces it uses, and it’s baudrate. These parameters must be stored so

that the software can call one function to manipulate different serial sensors using different pins and baudrates.

2.2 Defining the New Sensor Struct

Open the file `ssmSensor.h` and scroll to the commented definition of struct `NewSensor`. Replace this fake definition with a struct for your new sensor. Give it a name indicative of the physical sensor it is representing. Include all values that will need to be kept track of during the system's operation. Location of the new sensor struct shown in Figure 2.1 below.

Here are some items that may need to be stored in your struct:

- Pins used by the sensor
- Pointer to a driver class instance (for making measurement function calls)
- ADC channel (if using an external ADC)
- Power state & pin (if powering sensor via relay switch)

Next, add a new member to the union "Sensor" with the type of your new sensor struct (This union allows the "Channel" struct to store a struct for any sensor type). Finally, add a new value to the enum "SensorType" pertaining to your new sensor type. This will be used by the "Channel" struct to indicate what kind of sensor it's using.

```
87
88 //typedef struct NewSensor {
89 // int param1;
90 // char param2;
91 //};
92
93 /* Set this accordingly to reflect the kind of sensor used by a given channel */
94 typedef union Sensor {
95     SerialSensor ser;          /* Assigned if Sensor uses serial */
96     I2CSensor i2c;            /* Assigned if Sensor uses I2C */
97     SPISensor spi;           /* Assigned if Sensor uses SPI */
98     ChildSSM ssm;            /* Assigned if channel connects to another SSM */
99     // NewSensor new;
100 };
101
102 /* Indicates the kind of sensor used by a given channel */
103 typedef enum SensorType {
104     SENSOR_SERIAL,           /* Channel has a serial sensor */
105     SENSOR_I2C,              /* Channel has an I2C sensor */
106     SENSOR_SPI,              /* Channel has an SPI sensor */
107     SENSOR_SSM,              /* Channel has another SSM */
108     // SENSOR_NEW,
109     SENSOR_INVALID           /* Set if config file indicates unknown sensor type */
110 };
111
```

Figure 2.1: Commented sections indicate where to add code for a new sensor struct.

3. Writing Measurement Function

Open file **ssmSensor.cpp** and define a new measurement function with the following function prototype:

```
static int measureNewSensor(NewSensor *sensor,  
                           char *buf,  
                           int *len);
```

It's important that your new function has the same parameter and return types (except replace "NewSensor" with the name of your new sensor struct).

Design this function to gather one datapoint from your sensor. If your sensor uses a third party driver, this should be fairly simple. Ultimately, you want the data to be in string format for saving and serial transmission. If a conversion from float to char* is needed, use `dtostrf()` or another conversion function.

Make sure to add your new function prototype to **ssmSensor.h**.

Next, scroll to the function `measureSensor()` (shown in Figure 3.1 below) and add a case to its switch statement that calls your new measurement function if the sensor type is that of your new sensor. This generic function chooses which measurement function to use based on the sensor type being measured from.

```

81 |
82 | /*
83 | * Function: measureSensor
84 | * Generic measurement function. Takes a given sensor and calls
85 | * the appropriate measurement function based on its sensor type.
86 | * -----
87 | * *channel: Channel to measure from.
88 | * *buf:     Buffer to write stringified measurement data to.
89 | * *len:     Integer to write string length to.
90 | */
91 | int measureSensor(Channel *channel, char *buf, int *len) {
92 |     switch(channel->type) {
93 |         case SENSOR_SERIAL:
94 |             if(channel->sensor.ser.type == SERCOM_HARDWARE)
95 |                 return measureHardwareSerialSensor( &(channel->sensor.ser), buf, len);
96 |             //if(channel->sensor.ser.type == SERCOM_SOFTWARE)
97 |             // return measureSoftwareSerialSensor( &(channel->sensor.ser), buf, len);
98 |             break;
99 |         case SENSOR_I2C:
100 |             return measureI2CSensor( &(channel->sensor.i2c), buf, len);
101 |             break;
102 |         case SENSOR_SPI:
103 |             return measureSPISensor( &(channel->sensor.spi), buf, len);
104 |             break;
105 |         //case SENSOR_NEW:
106 |         // return measureNewSensor( &(channel->sensor.new), buf, len);
107 |         // break;
108 |     }
109 |
110 |     return -1;
111 | }
112 |

```

Figure 3.1: The function `measureSensor()` contains a commented out space for a call to a new measurement function.

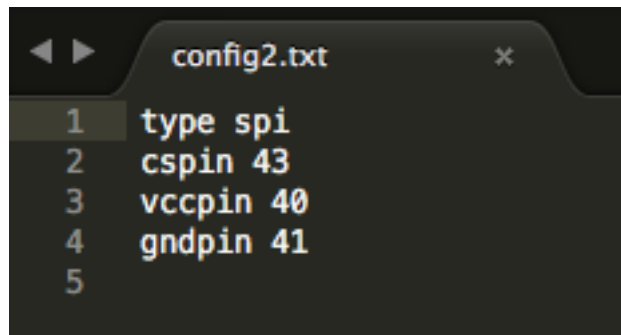
4. Writing configuration functions

4.1 Background

Upon startup, the Sensor Module reads a series of config files stored on it's SD card. The contents of these config files list information about the physical sensors on each channel (such as sensor type, pins, etc.). Much of this information must be transferred to the SSM struct's Channels upon startup for efficient access during operation.

There will be one config file for each channel that will be utilized during operation. When the config file is discovered by the system, it will be searched for specific keywords and the values associated with those keywords will be read and stored in the SSM struct.

In every config file, the first keyword will be "type". This will be followed by the type of sensor that this config file is describing. Next will be a series of keywords pertaining to that specific type of sensor and their respective values. An example config file is shown in Figure 4.1 below.



```
config2.txt
1 type spi
2 cspin 43
3 vccpin 40
4 gndpin 41
5
```

Figure 4.1. Config file for an SPI sensor listing the keywords `cspin`, `vccpin`, and `gndpin`, and their corresponding values.

4.2 Determining Config File Keywords

There must be one config file keyword for each parameter in your new sensor struct. The values associated with these keywords will be read and mapped to the struct's parameters upon startup. Decide upon a descriptive set of keywords to use in the config files for your new sensor.

4.2 Writing Configuration Functions

Once the config file keywords have been decided upon, open the file `ssmConfig.cpp`. Scroll to the section labeled "CONFIGURE NEW SENSOR". Define a new function in this section with the following prototype:

```
static void initNewSensor(Channel *channel, File *configfile)
```

Change "NewSensor" to the name of your sensor. The function will take two parameters. The first is of type `Channel*`, and refers to the channel that this sensor will be used by. It is assumed that the sensor type of this channel has already been determined to be your new sensor type. The second parameter is a `File` pointer to the config file containing this sensor's parameters. As already stated, it is assumed that if this function is called, the first line of the config file must indicate that the type is your new sensor type.

Design the function to search the config file for keywords pertaining to your sensor, read their values, and save those values to the new sensor struct within the `Channel` struct. To conveniently search the config files, the following functions are already defined in `ssmConfig.cpp`:

```
int getInt(File *configfile, const char *key)
```

Searches a config file for a given key, and converts its corresponding value to integer format.

```
int getChar(File *configfile, const char *key)
```

Searches a config file for a given key, and returns the first character of its corresponding value.

```
int getValueForKey(File *file, const char *key, char *value)
```

Searches a file for a given key, and copies the value to a char buffer.

Simple integer values like pin numbers can be easily parsed from config files with the `getInt()` function. However, unusual parameters may need to be read with `getValueForKey()` and processed further in code.

Additionally, if your new sensor uses a driver with a class instance to call functions, you will need to point to this instance in your sensor struct. If each sensor needs a unique instance of this driver class to operate, you should malloc an instance of this class in your config function and store the pointer in your struct. An example of this is shown in Figure 4.2 below.

```
195
196 /*
197  * Function: initSPISensor
198  * Initializes a channel to be a SPI sensor.
199  * -----
200  * *channel:    Channel to configure.
201  * *configfile: Config file to search.
202  */
203 static void initSPISensor(Channel *channel, File *configfile) {
204     Serial.println("SPI.");
205
206     /* Get pin numbers for SPI */
207     SPISensor spi;
208     spi.cspin = getInt( configfile, "cspin" );
209     spi.vccpin = getInt( configfile, "vccpin" );
210     spi.gndpin = getInt( configfile, "gndpin" );
211
212     /* Instantiate SPI probe object in heap - store pointer */
213     spi.probe = malloc(sizeof(SparkFunMAX31855k));
214     SparkFunMAX31855k probe(spi.cspin, spi.vccpin, spi.gndpin);
215     memcpy(spi.probe, &probe, sizeof(SparkFunMAX31855k));
216
217     channel->sensor.spi = spi;
218 }
219
```

Figure 4.2: Configuration function for a SPI sensor that allocates a new instance of a driver class for each sensor.

Be sure to set the member of the Sensor struct to the correct type, as shown in line 217 in Figure 4.2 above.

Finally, scroll down to the function `initChannel()`. In the switch statement, add a new case for your new sensor type and call your sensor configuration function. The switch statement in `initChannel()` is shown in Figure 4.3 below.

```

354  /* Configure channel based on what kind of sensor it is */
355  switch(channel->type) {
356      case SENSOR_SERIAL:
357          initSerialSensor(channel, &configfile);
358          break;
359      case SENSOR_I2C:
360          initI2CSensor(channel, &configfile);
361          break;
362      case SENSOR_SPI:
363          initSPISensor(channel, &configfile);
364          break;
365      case SENSOR_SSM:
366          initChildSSM(channel, &configfile);
367          break;
368      //case SENSOR_NEW:
369      //  initNewSensor(channel, &configfile);
370      //  break;
371      default:
372          channel->enabled = false;
373          Serial.println("Bad channel type");
374          break;
375  }

```

Figure 4.3: Switch statement within `initChannel()` with a commented case reserved for a new sensor type.

5. Conclusion

Now all of the necessary functions have been defined to integrate the new sensor. It may take a few iterations of testing and modification before the sensor is interfacing with the Sensor Module perfectly.