**In an effort to make JETTQManager less bloated the following functions were removed from the manager. They are however still present and can be accessed by their various managers.**

**Functions that are called to get individual LEDs were removed because they are more specific to manipulating the LEDs then the general workings of a SunSpot**

/\*\*

\* Turns on a single LED

\* @param i index of the LED to turn on

\*/

**public void turnLEDon(int i) {**

**this.ledm.setLED(i,true);**

**}**

/\*\*

\* Turns off a single LED

\* @param i index of the LED to turn off

\*/

**public void turnLEDoff(int i) {**

**this.ledm.setLED(i,false);**

**}**

/\*\*

\* Gets the color of a single LED

\* @param i index of the LED to get the color of  
 \* @return the color of the LED

\*/

**public** LEDColor getLEDColor(**int** i) {

**return** ledm.getColor(i);

}

**Functions Set Intensity, returning the LEDs, and making the LEDs display a number in binary were deemed unnecessary and more advanced then what the average program should do.**

/\*\*

\* Sets the intensity of all LEDs to the intensity value

\* Presets are Dim = 1, Moderate = 2, High = 3

\* **@param** intensity

\*/

**public** **void** setLEDIntensity(**int** intensity) {

**this**.ledm.setIntensity(intensity);

}

/\*\*

\* Returns an Enumeration of all the LEDs

\* **@return** an Enumeration of all the LEDs

\*/

**public** Enumeration getLEDs() {

**return** **this**.ledm.getLEDs();

}

/\*\*

\* Lights up LEDs like a binary clock!

\* The first LED represents 2^0, the second represents 2^1 and so on

\* **@param** num the number to be displayed in binary.

\* **@throws** Exception if the number is negative, or if there are not   
 \* enough LEDs

\*/

**public** **void** displayNumberAsBinary(**int** num) **throws** Exception {

**this**.ledm.displayNumberAsBinary(num);

}

**These accelerometer functions were removed because the AccelerometerManager’s update returns the accelerometer. The notification interval was deemed unnecessary to the basic functionality.**

/\*\*

\* Retrieves the Accelerometer from the Accelerometer Manager

\* **@return** IAceelerometer3D

\*/

**public** IAccelerometer3D getAccelerometer() {

**return** **this**.am.getAccelerometer();

}

/\*\*

\* Removes an observer from the Accelerometer observers

\* **@param** iao the IAccelerometerObserver to remove

\*/

**public** **void** removeAccelerometerObserver(IAccelerometerObserver iao) {

**this**.am.removeObserver(iao);

}

/\*\*

\* Sets the notification interval for the Accelerometer observers

\* **@param** ni int representing the milliseconds for the interval

\*/

**public** **void** setAccelerometerNotificationInterval(**int** ni) {

**this**.am.setNotificationInterval(ni);

}

**We decided that the stream radio would be too advanced and needed to be reserved to people that are willing to use the manager. To have in there.**

/\*\* STREAM RADIO \*\*/

/\*\*

\* Function setups a Radio Stream Connection to given Address on the given Port

\* **@param** ieeeAddress The Address of another Radio

\* **@param** portNumber The port to establish the connection on

\*\*/

**public** **void** radioStreamConnection(String ieeeAddress,**int** portNumber) {

JETTQManager.*decho*("Begin Stream Radio Connection Creation to: " + ieeeAddress + " on port " + portNumber);

**this**.rm.addRadioConnection(**new** StreamRadio(ieeeAddress,portNumber), portNumber);

}

/\*\*

\* Function retrieves the Data Input Stream from a RadioStreamConnection

\* **@return** DataInputStream

\*\*/

**public** DataInputStream getRadioDataInputStream(**int** portNumber) {

**return** **this**.rm.getDataInputStream(portNumber);

}

/\*\*

\* Function retrieves the Data Output Stream from a RadioStreamConnection

\* **@return** DataOutputStream

\*\*/

**public** DataOutputStream getRadioDataOutputStream(**int** portNumber) {

**return** **this**.rm.getDataOutputStream(portNumber);

}

**Broadcast was the only method we decided to keep in JETTQManager because the rest of the methods below required specifying more exact information about the type of connection. Since we only kept the broadcast once method in JETTQManager the stop sending method was unnecessary.**

/\*\* GRAM RADIO \*\*/

**public** **boolean** resetPort(**int** port) {

**return** **this**.rm.resetPort(port);

}

/\*\*

\* Broadcasts a message on a specific port at a specific interval.

\* **@param** msg Message object to send

\* **@param** interval Time in milliseconds to resend

\* **@param** port Port to send on.

\*/

**public** **void** sendMessage(Message msg, **int** interval, **int** port) {

JETTQManager.*decho*("Invoking Send Message on port: " + port);

**this**.rm.sendMessage(msg,interval,port);

}

/\*\*

\* Sends a message to a specific SPOT

\* **@param** to IEEE address of SPOT to send to

\* **@param** type Type of message being sent (user defined)

\* **@param** content String content for the message to be sent

\* **@param** interval Time in milliseconds to resend

\* **@param** port Port to send on

\*/

**public** **void** sendMessage(String to, **int** type, String content, **int** interval, **int** port) {

**this**.sendMessage(**new** Message(to,type,content), interval, port);

}

/\*\*

\* Removes a listener from the GramRadio

\* **@param** rgl IRadiogramObserver to remove

\* **@param** portNumber Port to stop listening on

\*/

**public** **void** removeRadiogramListener(IRadiogramObserver rgl, **int** portNumber) {

rm.removeRadiogramListener(rgl, portNumber);

}

/\*\*

\* Sets the retrieve interval for the listeners

\* **@param** interval Time in milliseconds to re-check

\* **@param** portNumber Port being listened on

\*/

**public** **void** setRetrieveInterval(**int** interval, **int** portNumber) {

**this**.rm.setRetrieveInterval(interval, portNumber);

}

/\*\*

\* Is the GramRadio sending on a specific port?

\* **@param** portNumber Port to check

\* **@return** true if the GramRadio is sending on the port, false otherwise

\*/

**public** **boolean** isSendingGramOn(**int** portNumber) {

**return** **this**.rm.isGramSendingOn(portNumber);

}

/\*\*

\* Is the GramRadio receiving on a specific port?

\* **@param** portNumber Port to check

\* **@return** true if the GramRadio is receiving on the port, false otherwise

\*/

**public** **boolean** isReceivingGramOn(**int** portNumber) {

**return** **this**.rm.isGramReceivingOn(portNumber);

}

/\*\*

\* Tells the GramRadio stop sending on the specific port

\* **@param** portNumber Port to stop sending on

\*/

**public** **void** stopSending(**int** portNumber){

**this**.rm.stopGramSending(portNumber);

}

/\*\*

\* Tells the GramRadio to stop receiving on the specific port

\* **@param** portNumber Port to stop receiving on

\*/

**public** **void** stopReceiving(**int** portNumber){

**this**.rm.stopGramReceiving(portNumber);

}

**These two methods were deemed unnecessary in JETTQManager because the update method returns the switch. The RemoveObserver was removed because we removed all the other removeObserver methods from JETTQManager**

/\*\*

\* Removes the observer for the switch

\* **@param** iao the switch observer to be removed

\*/

**public** **void** removeSwitchObserver(ISwitchObserver iso) {

sm.removeObserver(iso);

}

/\*\*

\* Is the specific switch pressed?

\* **@param** theswitch ISwitch object to check

\* **@return** true if the switch is pressed, false otherwise

\*/

**public** **boolean** isSwitchPressed(ISwitch theswitch)

{

**return** sm.isSwitchPressed(theswitch);

}

/\*\*

\* Is the specific switch pressed?

\* **@param** idx Index of the switch to check

\* **@return** true if the switch is pressed, false otherwise

\*/

**public** **boolean** isSwitchPressed(**int** idx)

{

**return** sm.isSwitchPressed(idx);

}

/\*\*

\* Gets a specific switch

\* **@param** idx Index of the switch to get

\* **@return** ISwitch object

\*/

**public** ISwitch getSwitch(**int** idx)

{

**return** sm.getSwitch(idx);

}