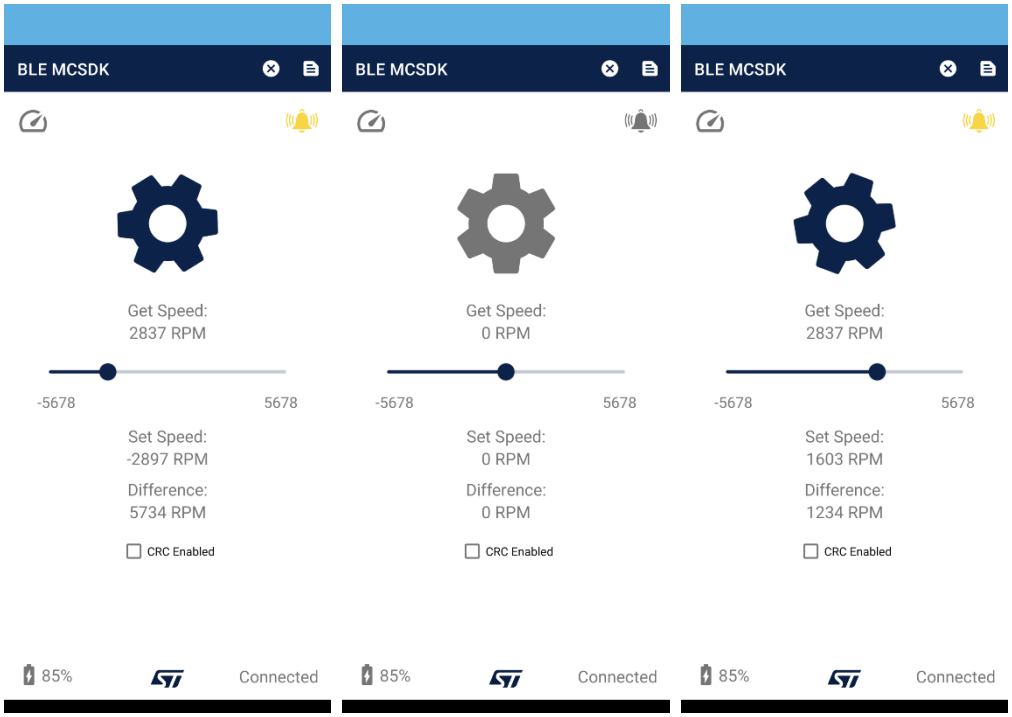
BLE-MCSDK
Android Application Documentation



Last Updated: 6/16/2022

App Name: BLE-MCSDK
App Version: 1.0.4
Package: com.stm.ble_mcsdk

Minimum Android SDK: 21(Android 5)
Target Android SDK: 31 (Android 12)

Summary:

The BLE-MCSDK is an application created by ST Microelectronics for mobile phones running Android. The app allows a user to connect to a ST Nucleo-WB55RG board with a motor attached to it and control the motor's speed (RPM) and direction through Bluetooth. The app will also request the minimum and maximum motor speed and periodically request the current motor speed every 100ms to display to the user. A live time-log is also available to view the application & Bluetooth activities.

Android Permissions:

- BLUETOOTH
- BLUETOOTH_ADMIN
- ACCESS_FINE_LOCATION
- ACCESS_COARSE_LOCATION
- BLUETOOTH_CONNECT
- BLUETOOTH_SCAN

Android Features:

- Bluetooth LE (Set to Required)
    - Hides the application on the Google Play Store from users without BLE capable devices

Dependencies:

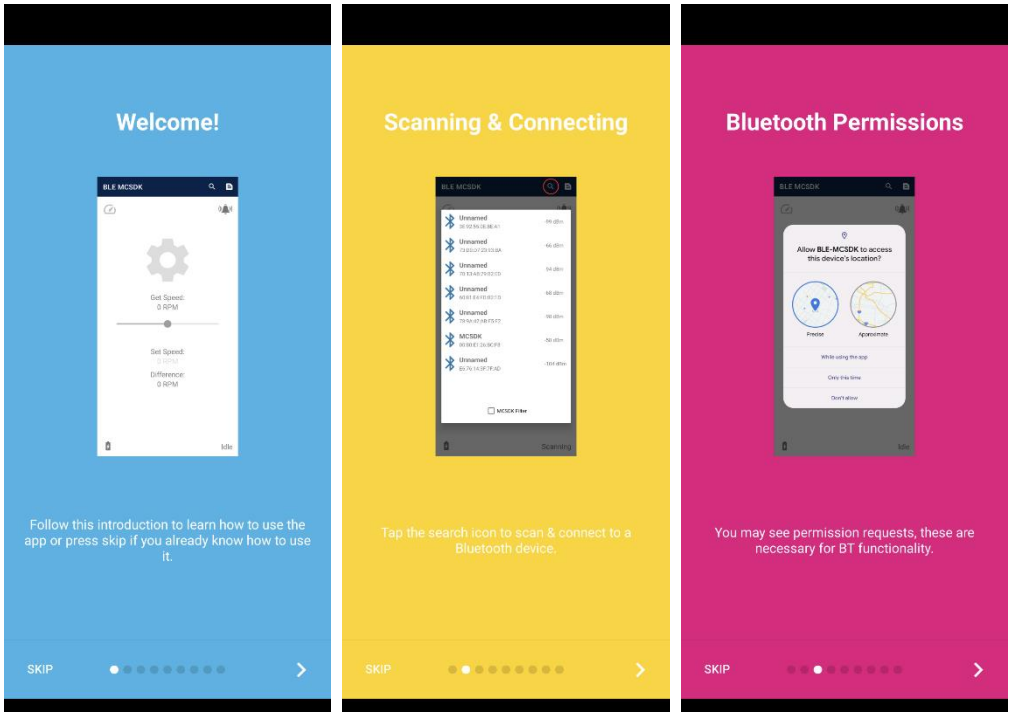
- Core-KTX
- AppCompat
- Material
- ConstraintLayout
- AppIntro
- Junit
- Espresso
- Timber
- Fragment
- Lifecycle-Runtime
- Lifecycle-ViewModel
- Lifecycle-LiveData

App Structure:

- Activities
    - Intro Activity
    - Main Activity
        - Main Interface
- BLE
    - BLE Manager
    - BLE Result
    - BLE Timeout Exception
- Databinding
    - Databinding
- Extensions

    - Byte Array & String
- Fragments
    - Log Fragment
        - Log Adapter
        - Log Interface
    - Reset Fragment
    - Scan Fragment
        - Scan Adapter
        - Scan Interface
    - MIN/MAX Speed Fragment
    - Set Speed Fragment
- Log
    - Log Manager
    - Log
- MCSDKApplication
- MCSDKViewModel

Intro Activity:



- Introduction Slides that Explain the App Functionality to First Time Users

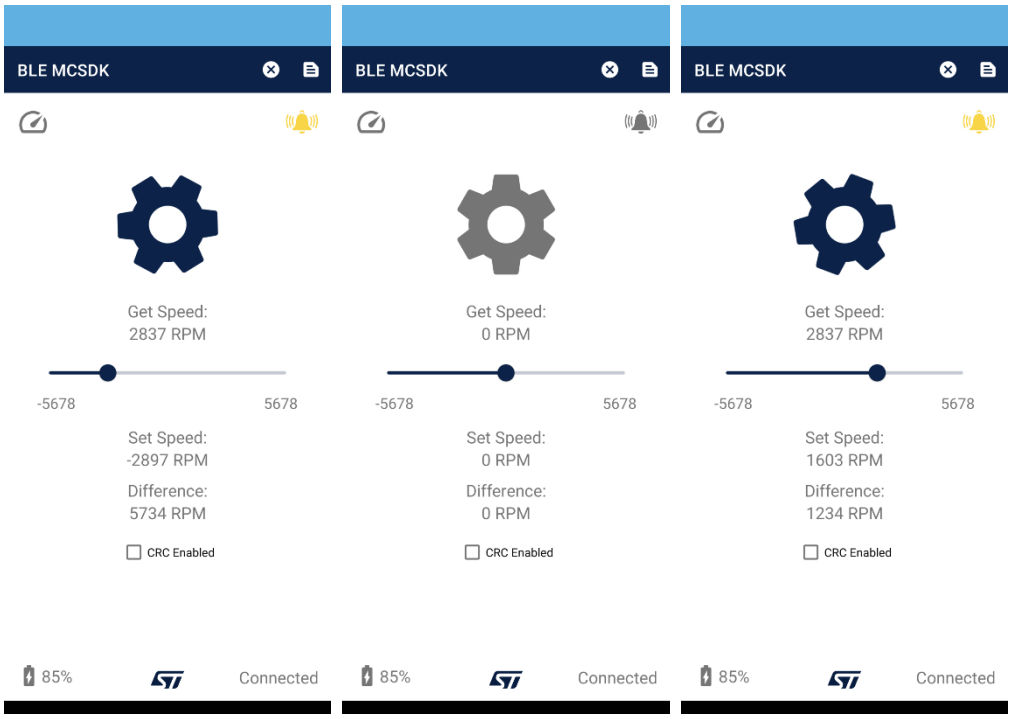
```
class IntroActivity : AppIntro() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        isSystemBackButtonLocked = true
        isColorTransitionsEnabled = true

        addSlide(AppIntroFragment.createInstance(
            title = "Welcome!",
            imageDrawable = R.drawable.ss_01,
            description = "Follow this introduction to learn how to use the app or press skip " +
                    "if you already know how to use it.",
            backgroundColorRes = R.color.st_light_blue
        ))
        addSlide(AppIntroFragment.createInstance(
            title = "Scanning & Connecting",
            imageDrawable = R.drawable.ss_02,
            description = "Tap the search icon to scan & connect to a Bluetooth device.",
            backgroundColorRes = R.color.st_yellow
        ))
```

- Uses the App Intro Library
- Adds Slides using the Library's App Intro Fragment

```
override fun onSkipPressed(currentFragment: Fragment?) {
    super.onSkipPressed(currentFragment)
    finish()
}

override fun onDonePressed(currentFragment: Fragment?) {
    super.onDonePressed(currentFragment)
    finish()
}
```

- Skip / Done Button
    - Goes to Main Activity

Main Activity:



- Main App Activity Where the User Can Connect to & Control the Motor
  - Application Starts Here

```
// Starts App Introduction if First Time Launching App
private fun appIntro() {
    if (sharedPrefs.getBoolean( key: "firstLaunch",  defValue: true)) {
        sharedPrefs.edit { putBoolean("firstLaunch", false) }
        val intent = Intent( packageContext: this, IntroActivity::class.java)
        startActivity(intent)
    }
}
```

- Starts App Introduction on First App Launch
  - Uses shared preferences to check if the first app launched
  - Goes to Intro Activity

```
// Prompt to Enable BT
override fun promptEnableBluetooth() {
    if(!bAdapter.isEnabled){
        val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
        ActivityCompat.startActivityForResult(
            activity: this, enableBtIntent, ENABLE_BLUETOOTH_REQUEST_CODE,  options: null
        )
    }
}
```

- Prompts User to Enable Bluetooth

- Called on activity resume

```
// Request Runtime Permissions (Based on Android Version)
private fun requestPermissions() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        requestMultiplePermissions.launch(arrayOf(
            Manifest.permission.ACCESS_FINE_LOCATION,
            Manifest.permission.BLUETOOTH_SCAN,
            Manifest.permission.BLUETOOTH_CONNECT
        ))
    } else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        requestMultiplePermissions.launch(arrayOf(
            Manifest.permission.ACCESS_FINE_LOCATION
        ))
    }
}

private val requestMultiplePermissions =
    registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) {}
```

- Requests Runtime Permissions from User
  - Android SDK 31 & up
    - ACCESS_FINE_LOCATION
    - BLUETOOTH_SCAN
    - BLUETOOTH_CONNECT
  - Android SDK 30 & below
    - ACCESS_FINE_LOCATION
  - Called on Search Item (search icon in toolbar) tap if permissions not granted

```
// Rerequest Permissions if Not Given by User (Limit 2)
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResults: IntArray
) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)

    if (BLEManager.hasPermissions( context: this)) {
        scanFragment.show(supportFragmentManager,  tag: "scanFragment")
    } else {
        requestPermissions()
    }
}
```

  - Opens Scan Fragment if permissions were granted
  - Prompts permission requests again if denied
    - Limited to 2 requests by Android
    - User will need to manually approve permissions in settings after 2 requests are made

```kotlin
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when(item.itemId) {
        R.id.searchItem -> {
            if (BLEManager.isConnected) {
                // Toggle Views & Set Speed Before Disconnecting
                toggleFunctionality( connected: false)
                BLEManager.disconnect()
            } else {
                if (!BLEManager.hasPermissions( context: this)) {
                    requestPermissions()
                    return false
                }

                scanFragment.show(supportFragmentManager,  tag: "scanFragment")
            }
        }
        R.id.logItem -> {
            logFragment.show(supportFragmentManager,  tag: "logFragment")
        }
    }

    return false
}
```

- Toolbar
    - Search Item (Magnifier Icon / Cancel Icon)
        - Icon changes when connected to BLE device
            - Not connected = magnifier icon
            - Connected = cancel icon
        - Connected
            - Views are disabled
            - Speed is reset
            - Disconnects from BLE device
        - Not Connected
            - Requests runtime permissions if not granted
                - Necessary for BLE functionality
            - Opens Scan Fragment
    - Log Item (Page Icon)
        - Opens Log Fragment
- Motor Icon (Cogwheel)

```
// Motor Image Clicked
fun onMotorClick() {
    with (viewModel) {  this: MCSDKViewModel
        isMotorOn = !isMotorOn

        if (isMotorOn) {
            BLEManager.scope.launch {  this: CoroutineScope
                // Turn Off Motor
                writeCharacteristic("030102" + calculateCRC( hexMessage: "030102"))
                LogManager.addLog( action: "Write Motor",  result: "OFF")

                delay( timeMillis: 200)

                // Turn On Motor
                writeCharacteristic("030101" + calculateCRC( hexMessage: "030101"))
                LogManager.addLog( action: "Write Motor",  result: "ON")

                delay( timeMillis: 200)

                // Write Get Max Speed Command
                writeCharacteristic( message: "02013F" + calculateCRC( hexMessage: "02013F"),  max: true)
                LogManager.addLog( action: "Write GET MAX",  result: "")

                delay( timeMillis: 200)

                // Write Speed Slider Value
                setSpeed(binding.speedSlider.value.toInt())

                delay( timeMillis: 200)

                // Start Get Speed Timer
                getSpeedMessage()
            }
            setMotorColor(R.attr.colorPrimary)
        } else {
            BLEManager.scope.launch {  this: CoroutineScope
                // Turn Off Motor
                writeCharacteristic("030102" + calculateCRC( hexMessage: "030102"))
                LogManager.addLog( action: "Write Motor",  result: "OFF")
            }
            setMotorColor(null)
        }

        toggleSpeed(isMotorOn)
    }
}
```

- On motor icon tap
  - Turn motor on
    - Sends motor off command
    - Sends motor on command
    - Sends get max speed command
      - Requests min & max motor speed values
    - Sends set speed command
      - Current speed value set by user
    - Starts get speed timer
    - Sets motor icon color to ST Dark Blue
  - Turn motor off
    - Sends motor off command
    - Resets motor icon color

- Calls toggleSpeed()

```
// Rotate Motor Animation
override fun rotateMotor(value: Int) {
    runOnUiThread {
        // Create Animation
        val rotate = RotateAnimation(
            if (value < 0) 360f else 0f,
            if (value < 0) 0f else 360f,
            Animation.RELATIVE_TO_SELF,  pivotXValue: 0.5f,
            Animation.RELATIVE_TO_SELF,  pivotYValue: 0.5f
        )
        rotate.interpolator = LinearInterpolator()
        rotate.repeatCount = -1

        // Calculate Animation Speed (Based on Slider Value)
        val valPercent = abs(value) / viewModel.maxSpeed.value!!.toFloat()
        val speed = 5000

        rotate.duration = when (valPercent) {
            0f -> 0
            1f -> 500
            else -> ((speed + 500) - (valPercent * speed)).toLong()
        }

        // Start Animation
        binding.motorImage.startAnimation(rotate)
    }
}
```

- Motor rotation animation
  - Creates a new rotate animation
  - Calculates animation speed & direction based on speed value set by the user
  - Starts the animation on the motor icon

```
@SuppressLint( ...value: "RestrictedApi")
private fun slider() {
    with(binding) {  this: ActivityMainBinding
        speedSlider.addOnSliderTouchListener(object: Slider.OnSliderTouchListener {
            override fun onStartTrackingTouch(slider: Slider) {}

            override fun onStopTrackingTouch(slider: Slider) {
                if (viewModel.isMotorOn) {
                    viewModel.setSpeed(slider.value.toInt())
                }
            }
        })

        speedSlider.addOnChangeListener(Slider.OnChangeListener { _, value, _ ->
            viewModel.setSpeed.value = value.toInt()
        })
    }
}
```

- Slider
  - When the user stops moving the slider
    - Speed value is set
    - Sends set speed command
  - While the user moves the slider
    - Set speed text value is updated

```
// Toggle Functionality based on BLE Connection
override fun toggleFunctionality(connected: Boolean) {
    runOnUiThread {
        with(binding) { this: ActivityMainBinding
            motorImage.isEnabled = connected

            if (connected) {
                // Connected to BLE Device
                searchItem?.setIcon(R.drawable.ic_cancel)
                batteryText.text = "${(80..90).random()}%"
            } else {
                // Not Connected to BLE Device
                searchItem?.setIcon(R.drawable.ic_search)
                setMotorColor(null)
                viewModel.isMotorOn = false
                toggleSpeed( toggle: false)
                batteryText.text = ""
            }
        }
    }
}
```

- Toggle Functionality
  - Toggles views, texts, and icons based on BLE connection
  - Connected
    - Motor icon enabled
    - Search icon in toolbar set to cancel icon
    - Random battery percentage set (80-90%)
  - Not connected
    - Motor icon disabled
    - Cancel icon in toolbar set to search icon
    - Reset motor icon color
    - Reset speed
    - Set battery text to empty (“”)

```
// Toggle Speed Values & Timer
private fun toggleSpeed(toggle: Boolean) {
    with (viewModel) {  this: MCSDKViewModel
        if (!toggle) {
            stopTimer()
            rotateMotor( value: 0)
            if (setSpeed.value != 0) setSpeed(0)
            if (getSpeed.value != 0) getSpeed.value = 0
        }
    }
}
```

- Toggle Speed
    - Toggles speed values, get speed timer, and motor animation
    - Toggle = false
        - Stops get speed timer
        - Turns off motor rotation animation
        - Sets set speed to 0
            - Sends speed value of 0 to the board
        - Sets get speed text to 0

```
// Set Bell Image Color
override fun setBellColor(color: Int) {
    runOnUiThread {
        binding.bellImage.drawable.setTint(
            ContextCompat.getColor(
                 context: this@MainActivity, color
            )
        )

        // Change Color Back After a Few Seconds
        Handler(Looper.getMainLooper()).postDelayed({
            binding.bellImage.drawable.setTint(
                ContextCompat.getColor(
                     context: this@MainActivity, R.color.grey
                )
            )
        }, delayMillis: 1000)
    }
}
```

- Bell Color
    - Sets bell icon color to given color

- Resets bell icon color after 1 second

BLE Manager:

- Standard Manager Object used in Multiple ST Applications
- Handles All BLE Operations

```
// List of BLE Scan Results
val scanResults = mutableListOf<ScanResult>()
```

- Scan Results
  - List of scanned BLE devices

```
fun startScan() {
    if (!isScanning) {
        scanResults.clear()
        scanAdapter.notifyDataSetChanged()

        bleScanner.startScan( filters: null, scanSettings, scanCallback)

        isScanning = true
        mainInterface?.updateStatus( status: "Scanning")
        LogManager.addLog( action: "Scan",  result: "Started")
        Timber.i( message: "BLE Scan Started")
    }
}

private fun stopScan() {
    if (isScanning) {
        bleScanner.stopScan(scanCallback)
        isScanning = false
        mainInterface?.updateStatus( status: "Scan Stopped")
        LogManager.addLog( action: "Scan",  result: "Stopped")
        Timber.i( message: "BLE Scan Stopped")
    }
}
```

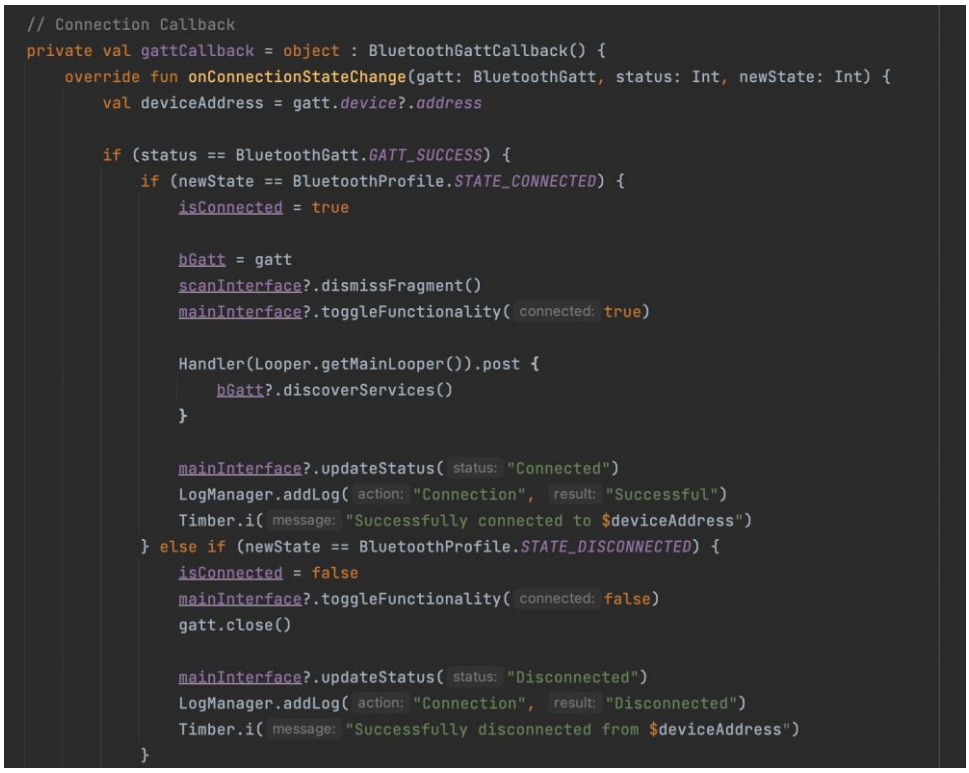
- Start / Stop BLE Scan
  - Updates status text on Main Activity

```
// Scan Result Callback
private val scanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        val indexQuery = scanResults.indexOfFirst { it.device.address == result.device.address }

        if (indexQuery != -1) { // Updates Existing Scan Result
            scanResults[indexQuery] = result
            scanAdapter.notifyItemChanged(indexQuery)
        } else { // Adds New Scan Result
            with(result.device) {  this: BluetoothDevice!
                Timber.i( message: "Found BLE device! Name: ${name ?: "Unnamed"}, address: $address")
            }

            // Filter by Device Name if Filter is On
            if (!filter || result.device.name == "MCSDK") {
                scanResults.add(result)
                scanAdapter.notifyItemInserted( position: scanResults.size - 1)
            }
        }
    }

    override fun onScanFailed(errorCode: Int) {
        mainInterface?.updateStatus( status: "Scan Failed")
        LogManager.addLog( action: "Scan",  result: "Failed")
        Timber.e( message: "BLE Scan Failed! Error Code: $errorCode")
    }
}
```

- Scan Call Back
    - Updates existing scan result if found in list
    - Adds new scan result to the list if not found
        - Result is added if result name matches filter or the filter is false
        - Filter is active (true) by default

```
// Connects to Scan Result Device
fun connect(result: ScanResult) {
    if (!isConnected) {
        stopScan()

        with(result.device) {  this: BluetoothDevice!
            connectGatt(app,  autoConnect: false, gattCallback)

            mainInterface?.updateStatus( status: "Connecting")
            LogManager.addLog( action: "Connection",  result: "Started")
            Timber.w( message: "Connecting to $address")
        }
    }
}

// Disconnects from Device
fun disconnect() {
    if (isConnected) bGatt?.disconnect()
}
```

- Connect / Disconnect from Device (Scan Result)
    - Updates status text on Main Activity



```
// Connection Callback
private val gattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        val deviceAddress = gatt.device?.address

        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                isConnected = true

                bGatt = gatt
                scanInterface?.dismissFragment()
                mainInterface?.toggleFunctionality( connected: true)

                Handler(Looper.getMainLooper()).post {
                    bGatt?.discoverServices()
                }

                mainInterface?.updateStatus( status: "Connected")
                LogManager.addLog( action: "Connection",  result: "Successful")
                Timber.i( message: "Successfully connected to $deviceAddress")
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                isConnected = false
                mainInterface?.toggleFunctionality( connected: false)
                gatt.close()

                mainInterface?.updateStatus( status: "Disconnected")
                LogManager.addLog( action: "Connection",  result: "Disconnected")
                Timber.i( message: "Successfully disconnected from $deviceAddress")
            }
```

- Connection Call Back
  - Connected
    - Scan fragment is dismissed
    - Main Activity views functionality enabled
    - Discovers connected device’s BLE services
  - Disconnected or connection failure
    - Main Activity views functionality disabled
  - Updates status text on Main Activity

```
override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
    with(gatt) {  this: BluetoothGatt
        Timber.i( message: "Discovered ${services.size} services for ${device.address}")
        printGattTable()

        scope.launch {  this: CoroutineScope
            try {
                requestMTU(GATT_MAX_MTU_SIZE)
            } catch (e: BLETimeoutException) {
                Timber.e( message: "BLE Timeout Error - MTU")
            }

            try {
                // Enable Write Response / Get Speed Notification
                val notifyCharacteristic = getCharacteristic(
                     serviceUUIDString: "0000fe40-cc7a-482a-984a-7f2ed5b3e58f",
                     characteristicUUIDString: "0000fe42-8e22-4541-9d4c-21edae82ed19"
                )
                enableNotifications(notifyCharacteristic)
            } catch (e: BLETimeoutException) {
                Timber.e( message: "BLE Timeout Error - Notify")
                disconnect()
            }
        }  ^with
    }
}
```

- Services Discovered
    - Requests larger MTU size
    - Enables notification for notify characteristic
        - Used for all BLE responses from the board
            - Get speed
            - Write commands

```
// Writes to a Characteristic
suspend fun writeCharacteristic(characteristic: BluetoothGattCharacteristic?, payload: ByteArray): BLEResult {
    val writeType = when {
        characteristic?.isWritable() == true -> BluetoothGattCharacteristic.WRITE_TYPE_DEFAULT
        characteristic?.isWritableWithoutResponse() == true -> {
            BluetoothGattCharacteristic.WRITE_TYPE_NO_RESPONSE
        }
        else -> error("Characteristic ${characteristic?.uuid} cannot be written to")
    }

    bGatt.let { gatt ->
        characteristic.writeType = writeType
        characteristic.value = payload
        gatt?.writeCharacteristic(characteristic)  ^let
    }
    return waitForResult(characteristic.uuid.toString())
}
```

- Write to Characteristic
    - Checks if given characteristic is writable or writable without a response and sets writeType accordingly
    - Sends the payload message to the characteristic

```
suspend fun enableNotifications(characteristic: BluetoothGattCharacteristic?): BLEResult? {
    val cccdUuid = UUID.fromString(CCC_DESCRIPTOR_UUID)
    val payload = when {
        characteristic?.isIndicatable() == true -> BluetoothGattDescriptor.ENABLE_INDICATION_VALUE
        characteristic?.isNotifiable() == true -> BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
        else -> {
            Timber.e( message: "${characteristic?.uuid} doesn't support notifications/indications")
            return null
        }
    }

    characteristic.getDescriptor(cccdUuid)?.let { cccDescriptor ->
        if (bGatt?.setCharacteristicNotification(characteristic,  enable: true) == false) {
            Timber.e( message: "setCharacteristicNotification failed for ${characteristic.uuid}")
            return null
        }
        return writeDescriptor(cccDescriptor, payload)
    } ?: Timber.e( message: "${characteristic.uuid} doesn't contain the CCC descriptor!")
    return null
}
```

- Enable Characteristic Notification
    - Checks if given characteristic is indicatable or notifiable
        - Sets payload value
    - Gets descriptor from given characteristic
    - Writes the payload to the descriptor

```
private suspend fun writeDescriptor(descriptor: BluetoothGattDescriptor, payload: ByteArray): BLEResult {
    bGatt?.let { gatt ->
        descriptor.value = payload
        gatt.writeDescriptor(descriptor)
        return waitForResult(descriptor.uuid.toString())
    } ?: error("Not connected to a BLE device!")
}
```

- Write to Descriptor
    - Writes payload to the given descriptor

Log Manager:

- Maintains a List of Logs for the Time-log

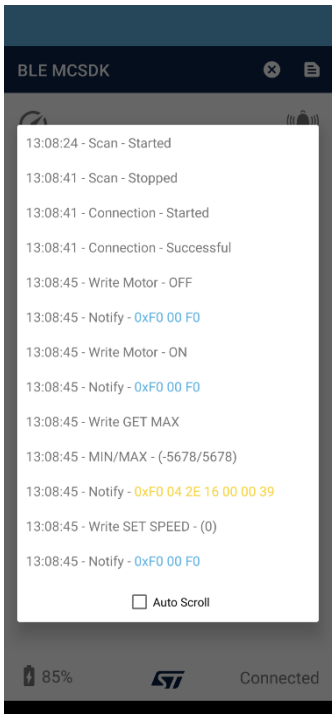
```
// Create & Add New Log
fun addLog(action: String, result: String, color: Int) {
    Handler(Looper.getMainLooper()).post {
        logs.add(Log(getTime(), action, result, color))
        logAdapter.notifyItemInserted( position: logs.size - 1)
        logInterface?.scrollRecyclerView()
    }
}

fun addLog(action: String, result: String) {
    addLog(action, result, R.color.grey)
}
```

- Add Log
    - Adds a log with the given information to the log list

```
// Get Current Time
private fun getTime(): String {
    with(Calendar.getInstance()) { this: Calendar
        val hour = timeFormat(get(Calendar.HOUR_OF_DAY))
        val min = timeFormat(get(Calendar.MINUTE))
        val sec = timeFormat(get(Calendar.SECOND))

        return "$hour:$min:$sec"
    }
}

private fun timeFormat(time: Int): String {
    return time.toString().padStart( length: 2,  padChar: '0')
}
```

- Get Time
    - Gets the current time from a calendar instance

Log Fragment & Adapter:

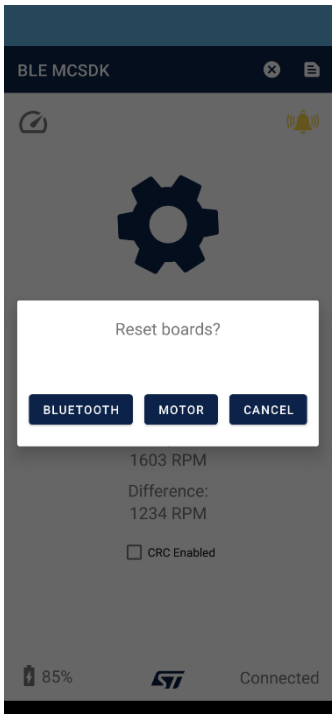


- Shows a Time-log of the Application and BLE Interactions

```
// Scroll to Bottom of Recycler View
override fun scrollRecyclerView() {
    if (viewCreated && binding.scrollCheckBox.isChecked) {
        val position = LogManager.logs.size - 1
        if (position != -1) {
            binding.logRecyclerView.smoothScrollToPosition(position)
        }
    }
}
```

- Auto Scroll
    - Scrolls to the last position in the logs RecyclerView
    - Controlled by the checkbox

Reset Fragment:



- Allows the User to Send Reset Commands to the Board

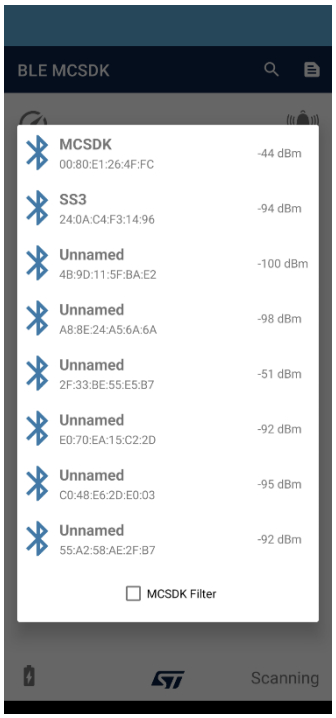
```
// Bluetooth Button Clicked
fun resetBlueTooth() {
    BLEManager.scope.launch {  this: CoroutineScope
        viewModel.writeCharacteristic("DEADBEEF42")
        LogManager.addLog( action: "Write RESET",  result: "Bluetooth")
    }
    dismissFragment()
}
```

  - Reset Bluetooth
    - Sends reset Bluetooth command
    - Adds a log to the time-log
    - Closes fragment

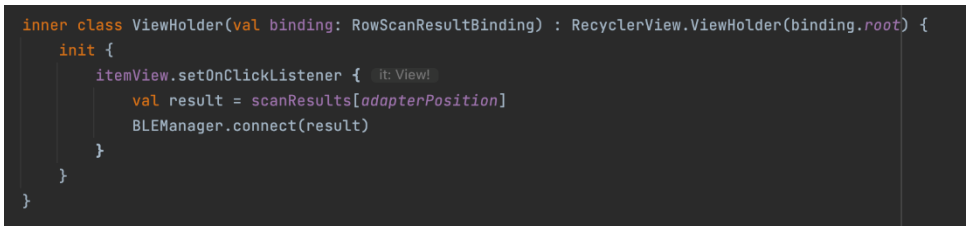
```
// Motor Button Clicked
fun resetMotor() {
    BLEManager.scope.launch {  this: CoroutineScope
        // Stop Get Speed Timer
        viewModel.stopTimer()

        delay( timeMillis: 200)

        // Turn Off Motor
        viewModel.writeCharacteristic("030102" + viewModel.calculateCRC( hexMessage: "030102"))
        LogManager.addLog( action: "Write Motor",  result: "OFF")

        delay( timeMillis: 200)

        // Reset Motor
        viewModel.writeCharacteristic("DEADBEEF4D")
        LogManager.addLog( action: "Write RESET",  result: "Motor")
    }
    dismissFragment()
}
```

    - Reset Motor
        - Stops get speed timer
        - Sends turn off motor command
        - Sends reset motor command
        - Adds logs to the time-log
        - Closes fragment
- Tap on the ST Icon to Show the Fragment
    - Only When Connected to a Device

Scan Fragment & Adapter:



- Shows a List of BLE Devices Allowing the User to Select and Connect to a Device



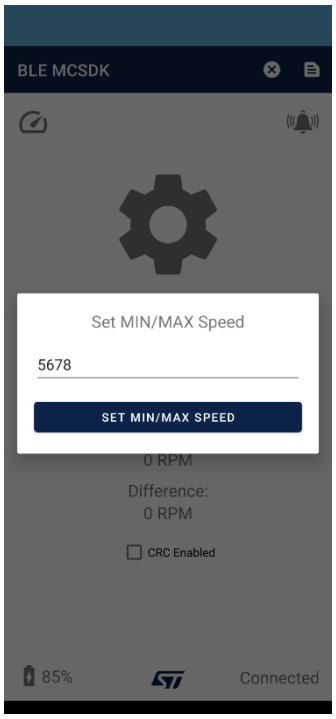
```
inner class ViewHolder(val binding: RowScanResultBinding) : RecyclerView.ViewHolder(binding.root) {
    init {
        itemView.setOnClickListener {  it: View!
            val result = scanResults[adapterPosition]
            BLEManager.connect(result)
        }
    }
}
```

- Item View (Row) Tap
    - Connect to selected device

```
@SuppressLint( ...value: "NotifyDataSetChanged")
fun filter(checkBox: CheckBox) {
    BLEManager.filter = checkBox.isChecked

    if (checkBox.isChecked) {
        resultsCopy.clear()
        resultsCopy.addAll(scanResults)
        BLEManager.scanResults.clear()

        for (result in resultsCopy) {
            if (result.device.name == "MCSDK") {
                BLEManager.scanResults.add(result)
            }
        }

        notifyDataSetChanged()
    }
}
```

- Filter
    - Adds list of scanned devices to a copy list
    - Checks each device in the copy list and adds each device back to the original list if they match the filter
        - Device name = “MCSDK”
    - Filter is true (checked) by default

MIN/MAX Speed Fragment:

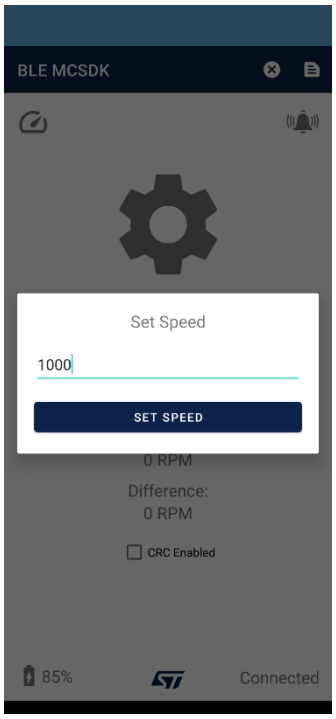


- Allows the User to Manually Enter the MIN/MAX Motor Speed
  - MIN speed will be the negative value of the user entered value
  - Updates the slider's MIN/MAX values

```
// Set Speed Button Clicked
fun onButtonClick() {
    viewModel.setMinMax(
        binding.speedEditText.text.toString().toInt()
    )
    dismiss()
}
```

- Button Tap
  - Sets MIN/MAX speed values as the user entered value
  - Closes fragment

Set Speed Fragment:



- Allows the User to More Precisely Enter a Speed Value
    - Updates the slider value

```
// Setup EditText Range
private fun setEditTextLimits() {
    with (binding) {  this: FragmentSetSpeedBinding
        speedEditText.doAfterTextChanged {  it: Editable?
            val text = speedEditText.text.toString()
            val max = viewModel.maxSpeed.value ?: 20000
            val min = max * -1

            if (text.isNotEmpty() && text != "-") {
                if (text.toInt() < min) speedEditText.setText(min.toString())
                if (text.toInt() > max) speedEditText.setText(max.toString())
            }
        }
    }
}
```

    - Allowed text input is limited between MIN & MAX values

```
// Set Speed Button Clicked
fun onButtonClick() {
    val speed = binding.speedEditText.text.toString().toInt()

    if (viewModel.isMotorOn) {
        viewModel.setSpeed(speed)
    } else {
        viewModel.setSpeed.value = speed
    }

    dismiss()
}
```

- Button Tap
    - Sends speed value to the motor if the motor is on
    - Updates the app's speed value if the motor is off
        - Value is sent to the motor when the motor icon is tapped to turn the motor on
    - Closes fragment

MCSDK Application:

```
class MCSDKApplication: Application() {
    override fun onCreate() {
        super.onCreate()
        app = this

        // Install Timber for Logging
        if (BuildConfig.DEBUG) {
            Timber.plant(Timber.DebugTree())
        }
    }

    companion object {
        lateinit var app: Application
    }
}
```

- Creates an Accessible Instance of the Application Class
- Installs Timber for Debug Logging if in Debug Build Configuration

MCSDK ViewModel:

- Additional BLE & Other App Specific Functionality that Does not Belong in the BLE Manager or Activities / Fragments

```
val setSpeed = MutableLiveData( value: 0)
val getSpeed = MutableLiveData( value: 0)
val maxSpeed = MutableLiveData( value: 2000)

var isMotorOn = false
var isCRCEnabled = false
```

- Live Data
  - Speed values used for the UI views and for sending and receiving speed values to and from the board

```
// Set Motor Speed
fun setSpeed(value: Int) {
    setSpeed.postValue(value)
    setSpeedMessage(value)
    if (isMotorOn) mainInterface.rotateMotor(value)
}
```

- Set Speed
  - Updates set speed value
  - Sends set speed to the board
  - Updates motor icon rotation animation if motor is on

```
// Set Motor MIN & MAX
fun setMinMax(max: Int) {
    // Keep Speed within Range
    if (setSpeed.value!! > max) {
        setSpeed(max)
    } else if (setSpeed.value!! < -max) {
        setSpeed(-max)
    }

    maxSpeed.postValue(max)
    LogManager.addLog( action: "MIN/MAX",  result: "(${max * -1}/${max})")
}
```

- Set MIN/MAX
  - Updates set speed value if outside of new range
  - Updates max speed value
  - Adds a log to the time-log

```
// Get MAX or Current Motor Speed
private fun getSpeed(hexValue: String) {
    val uSpeed = hexValue
        .filter { !it.isWhitespace() }
        .substring(6, 14)
        .hexToBigEndian()
        .toUInt( radix: 16)
    var speed = uSpeed.toInt()

    // Speed is Negative (Convert 2's Complement)
    if (uSpeed > Int.MAX_VALUE.toUInt()) {
        speed = ((UInt.MAX_VALUE - uSpeed) + 1u).toInt() * -1
    }

    if (maxCommand) {
        if (speed < 0) speed = Int.MAX_VALUE
        setMinMax(speed)
        maxCommand = false
    } else if (isMotorOn) {
        getSpeed.postValue(speed)
    }
}
```

- Get Speed
  - Converts a hexadecimal value sent from the connected board into an integer value
  - Converts value to a negative integer using 2’s complement if value is negative
  - If expecting a max value (get max speed command was sent), the speed value is set to max speed
  - If not expecting a max value and the motor is on, the speed value is set to get speed

```
// Write Set Speed Message from Slider
private fun setSpeedMessage(value: Int) {
    val command = "07"
    val size = "06"

    var speed = if (value < 0) {
        // Negative HEX (2's Complement)
        ((UInt.MAX_VALUE.toLong() + value) + 1).toString( radix: 16)
    } else {
        // Positive HEX
        value.toString( radix: 16)
    }
    speed = speed.padStart( length: 8,  padChar: '0').hexToLittleEndian()

    val rampDuration = "0000"

    val hexMessage = command + size + speed + rampDuration
    val crc = calculateCRC(hexMessage)

    BLEManager.scope.launch {  this: CoroutineScope
        writeCharacteristic(hexMessage + crc)
        LogManager.addLog( action: "Write SET SPEED",  result: "(${value})")
    }
}
```

- Set Speed Message
    - Converts a given integer value into hexadecimal
    - Converts value into a negative hexadecimal using 2’s complement if value is negative
    - Sends hexadecimal value (set speed) to the board

```
// Write Get Speed Message Every X Milliseconds
fun getSpeedMessage() {
    timer = Timer().apply {  this: Timer
        scheduleAtFixedRate(object : TimerTask() {
            override fun run() {
                BLEManager.scope.launch {  this: CoroutineScope
                    val message = if (isCRCEnabled) "02011E21" else "02011E"
                    writeCharacteristic(message)
                    LogManager.addLog( action: "Write GET SPEED",  result: "")
                }
            }
        }, delay: 0,  period: 100)
    }
    isTimerRunning = true
}
```

- Get Speed Message
    - Creates a new instance of a timer and starts it
    - Sends the get speed command to the board every 100 milliseconds

- Adds a log to the time-log every 100 milliseconds

```
suspend fun writeCharacteristic(message: String, max: Boolean): BLEResult? {
    // Write is a Get Max Command
    if (max) maxCommand = true

    // P2P Write Characteristic
    val characteristic = BLEManager.getCharacteristic(
        serviceUUIDString: "0000fe40-cc7a-482a-984a-7f2ed5b3e58f",
        characteristicUUIDString: "0000fe41-8e22-4541-9d4c-21edae82ed19"
    )
    val byteMessage = message.hexToByteArray()

    if (characteristic != null) {
        try {
            return BLEManager.writeCharacteristic(characteristic, byteMessage)
        } catch (e: BLETimeoutException) {
            Timber.e( message: "BLE Timeout Error - Write")
        }
    }
    return null
}

suspend fun writeCharacteristic(message: String): BLEResult? {
    return writeCharacteristic(message,  max: false)
}
```

- Write Characteristic
  - Called when writing a command to the connected board
  - Sets maxCommand variable to true if sending get max speed command
  - Converts given hexadecimal message to a byte array and writes it to the BLE characteristic

```
// Response from Write Command
fun writeResponse(hexValue: String, error: Boolean) {
    var mHexValue = if (error) "0xF0 XX" else hexValue
    var mError = error
    var mColor = R.color.st_light_blue
    var mEndMessage = ""

    // Check 2nd Hex Value to Determine Response
    with (mainInterface) {  this: MainInterface
        when (mHexValue.substring(5, 7)) {
            "00" -> setBellColor(R.color.st_light_blue)
            "01" -> {
                // Check 3rd Hex Value
                when (mHexValue.substring(8, 10).uppercase()) {
                    "03" -> {
                        mainInterface.showToast("Reading max speed is not allowed.")
                        mEndMessage = " (Read not allowed)"
                        maxCommand = false
                    }
                    "0A" -> {
                        mEndMessage = " (Bad CRC)"
                    }
                }

                mError = true
            }
            "04" -> {
                setBellColor(R.color.st_yellow)
                mColor = R.color.st_yellow
                getSpeed(mHexValue)
            }
            else -> {
                mError = true
            }
        }

        if (mError)  {
            mColor = R.color.st_pink
            setBellColor(R.color.st_pink)
        }
    }

    // Log Response
    mHexValue = hexValue.substring(0, 2) + hexValue.substring( startIndex: 2).uppercase()
    LogManager.addLog( action: "Notify",  result: mHexValue + mEndMessage, mColor)
}
```

- Write Response
    - Called when a response is received from the connected board (notification)
    - Checks hexadecimal value to determine app reaction to the response
    - Sets bell icon color based on response
        - Valid response = blue
        - Speed response = yellow
        - Error response = pink

    - Calls getSpeed() if speed response
    - Shows an error toast message for a specific hexadecimal value response
    - Adds a log to the time-log with the response information
        - Log color is changed based on response
- CRC
    - The CRC is the last byte (2 hexadecimal digits) in any command or response sent to or from the connected board
        - Used to check the hexadecimal message is correct
        - Sum of entire hexadecimal message constrained to a byte

```
fun calculateCRC(hexMessage: String): String {
    if (!isCRCEnabled) return ""

    // Calculate Sum of Hex Message
    val sum = hexMessage.hexCalculateSum()

    // Calculate Sum of HighByte + LowByte
    val crc = sum.padStart( length: 4,  padChar: '0').hexCalculateSum()

    // Return CRC (Truncated to a Byte)
    return crc.substring( startIndex: crc.length - 2)
}
```

    - Calculate CRC
        - Returns an empty string if CRC is disabled
        - Adds up the sum of each hexadecimal byte in the message (excluding the CRC if in the given message)
        - If the calculated sum is larger than a byte, the high byte and low byte are added together
        - CRC is truncated to a byte if still larger than a byte

```
// Check that CRC is Correct
fun checkCRC(hexMessage: String): Boolean {
    if (!isCRCEnabled) return true

    val message = hexMessage.filter { !it.isWhitespace() }

    val messageWithoutCRC = message.substring(2, message.length - 2)
    val crc = message.substring( startIndex: message.length - 2)

    return calculateCRC(messageWithoutCRC).uppercase() == crc.uppercase()
}
```

    - Check CRC
        - Returns true if CRC is disabled

- Checks that a given hexadecimal message's CRC is correct
- Removes white space from the given message
- Divides the given message into two separate messages
    - Message without the CRC
    - CRC
- Calculates a CRC for the message without the CRC and checks if it matches the given CRC

```
// Enables / Disables CRC Based on CRC Checkbox
fun toggleCRC(checkBox: CheckBox) {
    isCRCEnabled = checkBox.isChecked
}
```

- Toggle CRC
    - Enables or disables the CRC based on if the CRC checkbox is checked or not
        - Checked = Enabled | Not Checked = Disabled