```kotlin
package net.zeevox.nearow.output

import ...

class FitFileExporter(private val context: Context) {

   suspend fun exportTrackPoints(trackPoints: List<TrackPoint>): java.io.File {
       if (trackPoints.size <= 2)
           throw IllegalArgumentException("Too few trackPoints submitted to export to a file")

       val activity = createActivityFromTrackPoints(trackPoints)

       // create directory if not exists
       val directory = java.io.File(context.filesDir.path + "/exports")
       if (!directory.exists()) directory.mkdir()

       val file = java.io.File(directory, getFilenameForTimestamp(trackPoints.first().timestamp))
       writeMessagesToFile(activity, file)
       return file
   }

   private fun createActivityFromTrackPoints(trackPoints: List<TrackPoint>): List<Mesg> {

       val firstPoint = trackPoints.first()
       val lastPoint = trackPoints.last()

       val activityStartTime = DateTime(Date(firstPoint.timestamp))
       val activityEndTime = DateTime(Date(lastPoint.timestamp))

       val startLat = firstPoint.latitude?.let { decimalToGarmin(it) }
       val startLong = firstPoint.longitude?.let { decimalToGarmin(it) }
       val endLat = lastPoint.latitude?.let { decimalToGarmin(it) }
       val endLong = lastPoint.longitude?.let { decimalToGarmin(it) }

       val elapsedTime =
           ((activityEndTime.timestamp - activityStartTime.timestamp) / 1000).toFloat()

       return buildList {
           // Every FIT file MUST contain a File ID message
           add(getFileMetadata(activityStartTime))
           // A Device Info message is a BEST PRACTICE for FIT ACTIVITY files
           add(getDeviceInfo(activityStartTime))
           // Timer Events are a BEST PRACTICE for FIT ACTIVITY files
           add(createStartEvent(activityStartTime))
           // Create a RecordMesg for each TrackPoint and add it to the output queue
           trackPoints.mapTo(this, ::getRecordMesgForTrackPoint)
           // Every FIT file MUST contain at least one Lap message
           add(createLap(activityEndTime, elapsedTime, startLat, startLong, endLat, endLong))
           // Mark the activity as ended
           add(createEndEvent(activityEndTime))
           // Every FIT file MUST contain at least one Session message
           add(createSession(activityEndTime, elapsedTime, startLat, startLong))
           // Every FIT ACTIVITY file MUST contain EXACTLY one Activity message
           add(createActivityMesg(activityEndTime))
       }
   }
```

```kotlin
private suspend fun writeMessagesToFile(messages: List<Mesg?>, file: java.io.File) {
    // Create the output stream
    val encoder: FileEncoder =
        try {
            FileEncoder(file, Fit.ProtocolVersion.V2_0)
        } catch (e: FitRuntimeException) {
            Log.e(javaClass.simpleName, "Error opening file ${file.name}")
            e.printStackTrace()
            return
        }

    withContext(Dispatchers.IO) { for (message in messages) encoder.write(message) }

    // Close the output stream
    try {
        encoder.close()
    } catch (e: FitRuntimeException) {
        Log.e(javaClass.simpleName, "Error closing encode.")
        e.printStackTrace()
        return
    }

    Log.d(javaClass.simpleName, "Encoded FIT Activity file ${file.name}")
}

companion object {

    // The combination of manufacturer id, product id, and serial number should be unique.
    // When available, a non-random serial number should be used.
    private const val TRACKING_PRODUCT_ID: Int = 1
    private const val MANUFACTURER_ID: Int = Manufacturer.DEVELOPMENT
    private const val SOFTWARE_VERSION = BuildConfig.VERSION_CODE
    private const val SERIAL_NUMBER: Long = 2469834L

    /**
     * Garmin stores lat/long as integers. Each decimal degree represents 2^32 / 360 = 11930465
     * https://gis.stackexchange.com/a/368905
     */
    fun decimalToGarmin(pos: Double): Int = (pos * 11930465).toInt()

    private fun getRecordMesgForTrackPoint(trackPoint: TrackPoint) =
        RecordMesg().apply {
            timestamp = DateTime(Date(trackPoint.timestamp))
            speed = trackPoint.speed
            power = UnitConverter.speedToWatts(speed).toInt()
            cadence = trackPoint.strokeRate.toInt().toShort()
            trackPoint.latitude?.let { positionLat = decimalToGarmin(it) }
            trackPoint.longitude?.let { positionLong = decimalToGarmin(it) }
        }

    private fun createStartEvent(start: DateTime): EventMesg =
        EventMesg().apply {
            timestamp = start
            event = Event.TIMER
            eventType = EventType.START
            timerTrigger = TimerTrigger.MANUAL
            eventGroup = 0
        }
    private fun createEndEvent(end: DateTime): EventMesg =
        EventMesg().apply {
            timestamp = end
            event = Event.TIMER
            eventType = EventType.STOP
            timerTrigger = TimerTrigger.MANUAL
            eventGroup = 0
        }

    private fun createLap(
        lapStartTime: DateTime,
        elapsedTime: Float,
        _startPositionLat: Int? = null,
        _startPositionLong: Int? = null,
        _endPositionLat: Int? = null,
        _endPositionLong: Int? = null,
    ): LapMesg =
        LapMesg().apply {
            messageIndex = 0
            startTime = lapStartTime
            timestamp = lapStartTime

            totalElapsedTime = elapsedTime
            totalTimerTime = elapsedTime

            _startPositionLat?.let { startPositionLat = it }
            _startPositionLong?.let { startPositionLong = it }
            _endPositionLat?.let { endPositionLat = it }
            _endPositionLong?.let { endPositionLong = it }

            event = Event.LAP
            eventType = EventType.STOP
            lapTrigger = LapTrigger.MANUAL
            sport = Sport.ROWING
            subSport = SubSport.GENERIC
        }

    private fun getDeviceInfo(mesgTimestamp: DateTime): DeviceInfoMesg =
        DeviceInfoMesg().apply {
            deviceIndex = DeviceIndex.CREATOR
            manufacturer = MANUFACTURER_ID
            product = TRACKING_PRODUCT_ID
            serialNumber = SERIAL_NUMBER
            softwareVersion = SOFTWARE_VERSION.toFloat()
            timestamp = mesgTimestamp
        }
        private fun createSession(
            activityStartTime: DateTime,
            elapsedTime: Float,
            _startPositionLat: Int? = null,
            _startPositionLong: Int? = null,
        ): SessionMesg =
            SessionMesg().apply {
                messageIndex = 0
                firstLapIndex = 0
                numLaps = 0

                startTime = activityStartTime
                timestamp = activityStartTime

                totalElapsedTime = elapsedTime
                totalTimerTime = elapsedTime

                _startPositionLat?.let { startPositionLat = it }
                _startPositionLong?.let { startPositionLong = it }

                sport = Sport.ROWING
                subSport = SubSport.GENERIC

                event = Event.SESSION
                eventType = EventType.STOP
            }

        private fun getFileMetadata(startTime: DateTime): FileIdMesg =
            FileIdMesg().apply {
                type = File.ACTIVITY
                manufacturer = MANUFACTURER_ID
                product = TRACKING_PRODUCT_ID
                timeCreated = startTime
                serialNumber = SERIAL_NUMBER
            }

        private fun createActivityMesg(activityStartTime: DateTime): ActivityMesg {
            val timeZone: TimeZone = TimeZone.getDefault()
            val timezoneOffset: Long = (timeZone.rawOffset + timeZone.dstSavings) / 1000L
            return ActivityMesg().apply {
                timestamp = activityStartTime
                numSessions = 1
                type = Activity.MANUAL
                event = Event.ACTIVITY
                eventType = EventType.STOP
                localTimestamp = activityStartTime.timestamp + timezoneOffset
                totalTimerTime =
                    (activityStartTime.timestamp - activityStartTime.timestamp).toFloat()
            }
        }

        private fun getFilenameForTimestamp(timestamp: Long): String {
            // Create a DateFormatter object for displaying date in specified format.
            val formatter = SimpleDateFormat("yyyy-MM-dd-HH-mm-ss", Locale.UK)

            // Create a calendar object that will convert the date and time value in milliseconds to
            // date.
            val calendar = Calendar.getInstance().apply { timeInMillis = timestamp }

            return "Nero-${formatter.format(calendar.time)}.fit"
        }
    }
}
```