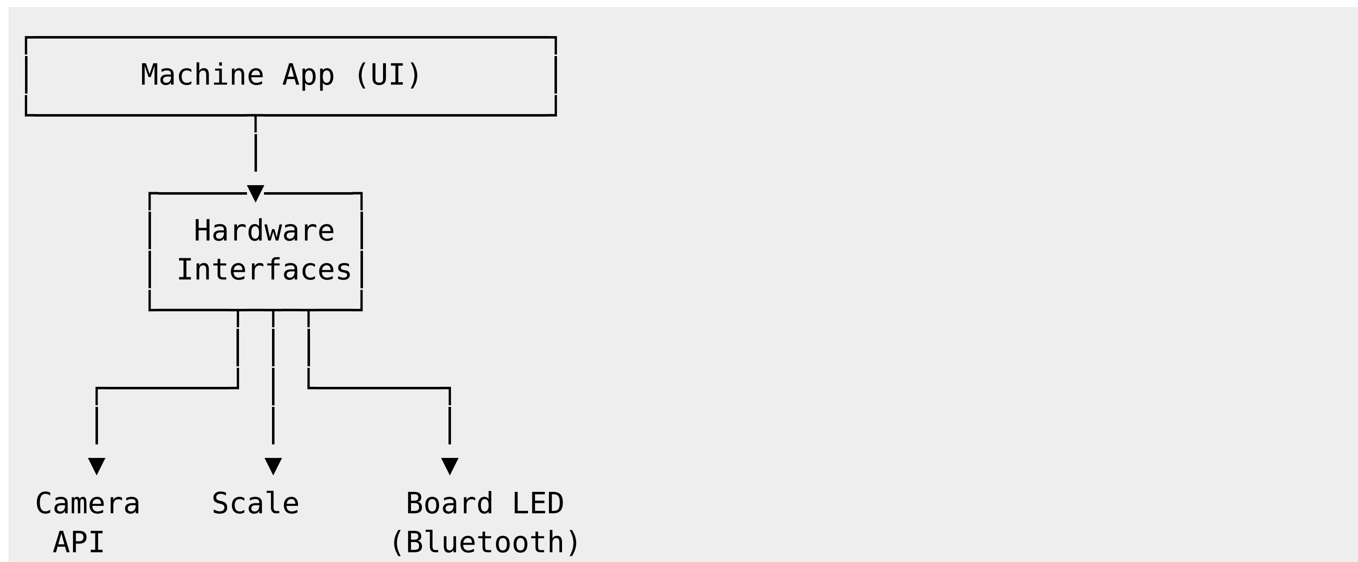
Integración de Hardware en PAE

Visión general

Machine integra varios componentes de hardware para la captura y procesamiento de entregas.



Camera2 API

Permisos

```
<!-- AndroidManifest.xml -->
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" android:required="true"
/>
```

Captura de imagen

Ubicación: `MachineDomain/src/…/hardware/Camera2Service.kt`

```
class Camera2Service(private val context: Context) {
    private lateinit var cameraManager: CameraManager
    private var cameraDevice: CameraDevice? = null
    private var captureSession: CameraCaptureSession? = null

    // Permisos en runtime
    fun requestCameraPermission(): Boolean {
        val permission = Manifest.permission.CAMERA
        return if (ContextCompat.checkSelfPermission(context, permission)
            != PackageManager.PERMISSION_GRANTED) {
```

```
            // Solicitar permisos
            false
        } else {
            true
        }
    }

    suspend fun takeSelfie(): Bitmap? = suspendCancellableCoroutine {
continuation ->
        try {
            cameraManager = context.getSystemService(Context.CAMERA_SERVICE)
as CameraManager

            // Enumerar cámaras disponibles
            val cameraIds = cameraManager.cameraIdList
            val frontCameraId = cameraIds.firstOrNull { id ->
                val characteristics =
cameraManager.getCameraCharacteristics(id)
                characteristics.get(CameraCharacteristics.LENS_FACING) ==
                    CameraCharacteristics.LENS_FACING_FRONT
            }

            if (frontCameraId != null) {
                cameraManager.openCamera(frontCameraId, object :
CameraDevice.StateCallback() {
                    override fun onOpened(camera: CameraDevice) {
                        cameraDevice = camera
                        startCapture(continuation)
                    }

                    override fun onDisconnected(camera: CameraDevice) {
                        camera.close()
                        continuation.resume(null)
                    }

                    override fun onError(camera: CameraDevice, error: Int) {
                        camera.close()
                        continuation.resumeWithException(
                            Exception("Camera error: $error")
                        )
                    }
                }, Handler(Looper.getMainLooper()))
            } else {
                continuation.resumeWithException(
                    Exception("No front camera found")
                )
            }
        } catch (e: SecurityException) {
            continuation.resumeWithException(e)
        }
```

```
    }

    private fun startCapture(continuation: Continuation<Bitmap?>) {
        val imageCaptureTarget = object :
ImageReader.OnImageAvailableListener {
            override fun onImageAvailable(reader: ImageReader?) {
                reader?.acquireNextImage()?.use { image ->
                    val planes = image.planes
                    val buffer = planes[0].buffer
                    val pixelStride = planes[0].pixelStride

                    // Convertir a Bitmap
                    buffer.rewind()
                    val data = ByteArray(buffer.remaining())
                    buffer.get(data)

                    val bitmap = BitmapFactory.decodeByteArray(data, 0,
data.size)
                    continuation.resume(bitmap)
                }
            }
        }

        val imageReader = ImageReader.newInstance(1280, 720,
ImageFormat.JPEG, 1)
        imageReader.setOnImageAvailableListener(imageCaptureTarget,
Handler())

        val surface = imageReader.surface
        val requestBuilder = cameraDevice?.createCaptureRequest(
            CameraDevice.TEMPLATE_STILL_CAPTURE
        )
        requestBuilder?.addTarget(surface)

        cameraDevice?.createCaptureSession(
            listOf(surface),
            object : CameraCaptureSession.StateCallback() {
                override fun onConfigured(session: CameraCaptureSession) {
                    captureSession = session
                    try {
                        session.capture(
                            requestBuilder?.build()!!,
                            null,
                            Handler()
                        )
                    } catch (e: CameraAccessException) {
                        continuation.resumeWithException(e)
                    }
                }

                override fun onConfigureFailed(session:
```

```
CameraCaptureSession) {
                    continuation.resumeWithException(
                        Exception("Camera capture session failed")
                    )
                }
            },
            Handler()
        )
    }

    fun close() {
        captureSession?.close()
        cameraDevice?.close()
    }
}
```

Uso en StateManager

```
// CaptureFaceState.kt
override suspend fun run(next: () -> Unit, retry: (String) -> Unit) {
    try {
        val bitmap = camera2Service.takeSelfie()

        if (bitmap != null) {
            // Procesar imagen
            stateData.facePhoto = bitmap
            next()
        } else {
            retry("No se pudo capturar foto")
        }
    } catch (e: Exception) {
        retry("Error capturando foto: ${e.message}")
    }
}
```

Balanza (Scale) - Bluetooth

Permisos

```
<!-- AndroidManifest.xml -->
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
```

Interfaz de balanza

Ubicación: `MachineDomain/src/…/hardware/ScaleManager.kt`

```
interface Scale {
    suspend fun getWeight(): Float?
    suspend fun tare()
    fun addEventListener(listener: ScaleEventListener)
    fun close()
}

interface ScaleEventListener {
    fun onWeightChanged(weight: Float)
    fun onConnectionStatusChanged(connected: Boolean)
    fun onError(error: String)
}
```

Implementación Bluetooth

```
class BluetoothScaleManager(
    private val context: Context,
    private val deviceMacAddress: String
) : Scale {
    private var bluetoothAdapter: BluetoothAdapter? = null
    private var bluetoothSocket: BluetoothSocket? = null
    private var listeners = mutableListOf<ScaleEventListener>()

    suspend fun connect(): Boolean = suspendCancellableCoroutine {
continuation ->
        try {
            bluetoothAdapter = BluetoothAdapter.getDefaultAdapter()

            val device = bluetoothAdapter?.getRemoteDevice(deviceMacAddress)
                ?: return@suspendCancellableCoroutine
continuation.resume(false)

            bluetoothSocket = device.createRfcommSocketToServiceRecord(
                UUID.fromString("00001101-0000-1000-8000-00805F9B34FB") //
SPP UUID
            )

            bluetoothSocket?.connect()

            // Iniciar lectura
            launchWeightReader()

            continuation.resume(true)
        } catch (e: Exception) {
            continuation.resume(false)
```

```
        }
    }

    private fun launchWeightReader() {
        val inputStream = bluetoothSocket?.inputStream ?: return

        Thread {
            val buffer = ByteArray(1024)

            try {
                while (true) {
                    val bytes = inputStream.read(buffer)
                    val data = String(buffer, 0, bytes)

                    // Parse: típicamente formato "12.5 kg\r\n"
                    val weight = parseWeight(data)
                    if (weight != null) {
                        listeners.forEach { it.onWeightChanged(weight) }
                    }
                }
            } catch (e: IOException) {
                listeners.forEach { it.onError("Balanza desconectada") }
                listeners.forEach { it.onConnectionStatusChanged(false) }
            }
        }.start()
    }

    private fun parseWeight(data: String): Float? {
        return try {
            // Ejemplo: "12.5 kg"
            val parts = data.trim().split(" ")
            parts[0].toFloat()
        } catch (e: Exception) {
            null
        }
    }

    override suspend fun getWeight(): Float? = suspendCancellableCoroutine {
continuation ->
        // Retorna el peso actual
        // (implementación depende del protocolo de la balanza)
    }

    override suspend fun tare() {
        bluetoothSocket?.outputStream?.apply {
            write("TARE\r\n".toByteArray())
            flush()
        }
    }
```

```
    override fun addEventListener(listener: ScaleEventListener) {
        listeners.add(listener)
    }

    override fun close() {
        bluetoothSocket?.close()
    }
}
```

Uso en StateManager

```
// WaitingForWeightState.kt
override suspend fun run(next: () -> Unit, retry: (String) -> Unit) {
    var weightReceived = false

    scaleManager.addEventListener(object : ScaleEventListener {
        override fun onWeightChanged(weight: Float) {
            if (weight > 0.5f && !weightReceived) { // peso mínimo
                weightReceived = true
                stateData.weight = weight
                next()
            }
        }

        override fun onConnectionStatusChanged(connected: Boolean) {
            if (!connected) {
                retry("Balanza desconectada")
            }
        }

        override fun onError(error: String) {
            retry("Error balanza: $error")
        }
    })
}
```

Indicadores LED - GPIO

Interfaz

```
interface BoardLed {
    fun turnOn(color: LedColor)
    fun turnOff()
    fun blink(color: LedColor, durationMs: Long)
    fun stopBlinking()
}
```

```
enum class LedColor {
    RED, GREEN, YELLOW
}
```

Implementación sencilla

```
class GpioLedManager : BoardLed {
    // Típicamente se comunica vía USB o serie

    override fun turnOn(color: LedColor) {
        val command = when (color) {
            LedColor.RED -> "LED_RED_ON"
            LedColor.GREEN -> "LED_GREEN_ON"
            LedColor.YELLOW -> "LED_YELLOW_ON"
        }
        sendCommand(command)
    }

    override fun turnOff() {
        sendCommand("LED_OFF")
    }

    override fun blink(color: LedColor, durationMs: Long) {
        val command = when (color) {
            LedColor.RED -> "LED_RED_BLINK,$durationMs"
            LedColor.GREEN -> "LED_GREEN_BLINK,$durationMs"
            LedColor.YELLOW -> "LED_YELLOW_BLINK,$durationMs"
        }
        sendCommand(command)
    }

    private fun sendCommand(command: String) {
        // Enviar vía puerto serie o USB
    }
}
```

Uso en StateManager

```
// GenerateEmbeddingState.kt
override suspend fun run(next: () -> Unit, retry: (String) -> Unit) {
    try {
        boardLed.turnOn(LedColor.YELLOW) // procesando

        val embedding = generateEmbedding(stateData.facePhoto)
        stateData.embedding = embedding

        boardLed.turnOn(LedColor.GREEN) // éxito
```

```
        next()
    } catch (e: Exception) {
        boardLed.blink(LedColor.RED, 500)
        retry("Error generando embedding: ${e.message}")
    }
}
```

Almacenamiento de fotos

Ubicación segura

```
class PhotoStorage(private val context: Context) {
    private val photosDir = File(context.filesDir, "photos")

    init {
        photosDir.mkdirs()
    }

    fun saveBeneficiaryPhoto(bitmap: Bitmap, beneficiaryId: Long): String {
        val filename =
"beneficiary_${beneficiaryId}_${System.currentTimeMillis()}.jpg"
        val file = File(photosDir, filename)

        FileOutputStream(file).use { fos ->
            bitmap.compress(Bitmap.CompressFormat.JPEG, 95, fos)
        }

        return file.absolutePath
    }

    fun saveDeliveryPhoto(bitmap: Bitmap, deliveryId: Long): String {
        val filename =
"delivery_${deliveryId}_${System.currentTimeMillis()}.jpg"
        val file = File(photosDir, filename)

        FileOutputStream(file).use { fos ->
            bitmap.compress(Bitmap.CompressFormat.JPEG, 90, fos)
        }

        return file.absolutePath
    }

    fun deletePhoto(path: String): Boolean {
        return File(path).delete()
    }

    fun getPhotoSize(path: String): Long {
        return File(path).length()
```

```
    }
}
```

Inicialización de hardware

```
// Machine app initialization

class HardwareInitializer(private val context: Context) {
    suspend fun initializeAll(): HardwareServices {
        return try {
            val camera2Service = Camera2Service(context)

            val scaleManager = BluetoothScaleManager(
                context,
                MAC_ADDRESS_SCALE
            ).apply {
                connect()
            }

            val boardLed = GpioLedManager()
            val photoStorage = PhotoStorage(context)

            HardwareServices(
                camera = camera2Service,
                scale = scaleManager,
                led = boardLed,
                photoStorage = photoStorage
            )
        } catch (e: Exception) {
            Logger.e("HW_INIT", "Error inicializando hardware:
${e.message}")
            throw e
        }
    }
}

data class HardwareServices(
    val camera: Camera2Service,
    val scale: Scale,
    val led: BoardLed,
    val photoStorage: PhotoStorage
)
```

Simulación para desarrollo

```
// Para testing sin hardware físico
```

```
class MockScale : Scale {
    override suspend fun getWeight(): Float? = Random.nextFloat() * 50

    override suspend fun tare() { /* no-op */ }

    override fun addEventListener(listener: ScaleEventListener) {
        // Simular cambio de peso
        Thread {
            Thread.sleep(2000)
            listener.onWeightChanged(12.5f)
        }.start()
    }

    override fun close() { /* no-op */ }
}

class MockCamera2Service : Camera2Service(context) {
    override suspend fun takeSelfie(): Bitmap? {
        // Retornar imagen de prueba
        return BitmapFactory.decodeResource(
            context.resources,
            R.drawable.test_face
        )
    }
}

class MockBoardLed : BoardLed {
    override fun turnOn(color: LedColor) {
        println("LED: ${color.name} ON")
    }

    override fun turnOff() {
        println("LED: OFF")
    }

    override fun blink(color: LedColor, durationMs: Long) {
        println("LED: ${color.name} BLINK $durationMs ms")
    }

    override fun stopBlinking() {
        println("LED: STOP BLINK")
    }
}
```

Manejo de desconexiones

```
class HardwareMonitor(private val hardware: HardwareServices) {
    fun monitorAll() {
```

```
        // Monitorear balanza
        hardware.scale.addEventListener(object : ScaleEventListener {
            override fun onConnectionStatusChanged(connected: Boolean) {
                if (!connected) {
                    handleScaleDisconnection()
                }
            }

            override fun onWeightChanged(weight: Float) { }
            override fun onError(error: String) {
                handleScaleError(error)
            }
        })
    }

    private fun handleScaleDisconnection() {
        Logger.w("MONITOR", "Balanza desconectada")
        // Notificar a UI
        // Retry conexión
    }

    private fun handleScaleError(error: String) {
        Logger.e("MONITOR", "Error en balanza: $error")
    }
}
```

Calibración de hardware

```
class HardwareCalibration(private val context: Context) {
    suspend fun calibrateScale(scale: Scale) {
        // 1. Tare (poner a cero)
        scale.tare()
        delay(1000)

        // 2. Colocar peso conocido (ej: 1 kg)
        // 3. Leer peso
        val readWeight = scale.getWeight()

        // 4. Calcular factor de calibración
        val calibrationFactor = 1000.0f / (readWeight ?: 1f)

        // 5. Guardar
        saveSetting("scale_calibration_factor", calibrationFactor)
    }

    private fun saveSetting(key: String, value: Float) {
        val prefs = context.getSharedPreferences("hardware",
Context.MODE_PRIVATE)
```

```
        prefs.edit().putFloat(key, value).apply()
    }
}
```

Energía y battery saver

```
// Optimización para máquinas con batería limitada

class PowerManager(private val context: Context) {
    fun enableBatterySaverMode() {
        // Reducir frecuencia de polling
        // Apagar componentes no usados
        // Reducir resolución de cámara
        // Aumentar timeout de pantalla
    }

    fun disableBatterySaverMode() {
        // Restaurar configuración normal
    }

    fun getDeviceBatteryPercentage(): Int {
        val batteryManager =
context.getSystemService(BatteryManager::class.java)
        return
batteryManager?.getIntProperty(BatteryManager.BATTERY_PROPERTY_CHARGE_COUNTE
R) ?: 0
    }
}
```

From:
http://wiki.adacsc.co/ - **Wiki**

Permanent link:
http://wiki.adacsc.co/doku.php?id=ada:howto:sicoferp:factory:new-migracion-sicoferp:hardware-integration

Last update: **2026/04/07 20:02**