技 术 文 件

技术文件名称： 轻智能手表系统设计方案

技术文件编号：

版 本： V0.1

拟 制

审 核

会 签

标准化

批 准

海宁奕斯伟集成电路设计有限公司

修改记录

| 文件编号 | 版本号 | 拟制人/  修改人 | 拟制/修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 注1：每次更改归档文件（指归档到事业部或公司档案室的文件）时，需填写此表。  注2：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。 | | | | | |

目录

[1 背景介绍 8](#_Toc90539856)

[1.1 轻智能手表背景介绍 8](#_Toc90539857)

[1.2 轻智能手表功能描述 8](#_Toc90539858)

[2 主要开发工作内容概述 9](#_Toc90539859)

[2.1 开发工作主要内容 9](#_Toc90539860)

[2.2 开发工作的规划 10](#_Toc90539861)

[3 toolchain工具链 10](#_Toc90539862)

[3.1 gcc工具链 10](#_Toc90539863)

[3.2 IDE 11](#_Toc90539864)

[3.2.1 IDE整体功能与框架 11](#_Toc90539865)

[3.2.2 Native程序的调试 12](#_Toc90539866)

[3.2.3 JavaScript应用程序的调试 12](#_Toc90539867)

[4 emulator 13](#_Toc90539868)

[5 内核层 14](#_Toc90539869)

[5.1 设备驱动 14](#_Toc90539870)

[5.2 协议栈 15](#_Toc90539871)

[5.3 功耗管理 15](#_Toc90539872)

[5.4 文件系统 15](#_Toc90539873)

[5.4.1 Nuttx VFS 16](#_Toc90539874)

[5.4.2 Littlefs文件系统 16](#_Toc90539875)

[5.4.3 Romfs文件系统 16](#_Toc90539876)

[5.5 基础库 17](#_Toc90539877)

[5.6 调度系统 17](#_Toc90539878)

[5.7 SDK 18](#_Toc90539879)

[5.7.1 处理器部分 18](#_Toc90539880)

[5.7.2 SOC和板级部分 18](#_Toc90539881)

[5.7.3 外设部分 18](#_Toc90539882)

[5.7.4 HAL 18](#_Toc90539883)

[6 编译框架 19](#_Toc90539884)

[6.1 编译架构与构建 19](#_Toc90539885)

[6.2 文件系统与flash分区 20](#_Toc90539886)

[6.2.1 文件系统生成 20](#_Toc90539887)

[6.2.2 Flash分区 20](#_Toc90539888)

[7 JS引擎和框架 21](#_Toc90539889)

[7.1 JS引擎概述及原理 21](#_Toc90539890)

[7.2 JS框架分层设计 21](#_Toc90539891)

[7.2.1 JS Adaptation层 22](#_Toc90539892)

[7.2.2 JS Runtime层 23](#_Toc90539893)

[7.2.3 JS 引擎内置功能 23](#_Toc90539894)

[7.2.4 JS SDK层 24](#_Toc90539895)

[7.2.5 应用开发层 24](#_Toc90539896)

[8 显示子系统 24](#_Toc90539897)

[8.1 GUI库 25](#_Toc90539898)

[9 摄像头子系统 26](#_Toc90539899)

[10 OTA子系统 26](#_Toc90539900)

[11 音频子系统 26](#_Toc90539901)

[12 电话子系统 26](#_Toc90539902)

[13 sensor子系统 26](#_Toc90539903)

[14 安全子系统 26](#_Toc90539904)

[15 扩展子系统 27](#_Toc90539905)

[15.1 蓝牙子系统 27](#_Toc90539906)

[15.2 AI子系统 27](#_Toc90539907)

[15.3 Cloud子系统 27](#_Toc90539908)

[16 术语、定义和缩略语 27](#_Toc90539909)

[16.1 术语、定义 27](#_Toc90539910)

[17 参考文献 27](#_Toc90539911)

[18 问题 27](#_Toc90539912)

# 背景介绍

## 轻智能手表背景介绍

轻智能手表项目是和SC BU合作，开发一款面向轻智能手表市场的芯片。其架构大体可以分为AP部分和CP部分（包括基带和sensor hub）。CP部分的驱动和调试由SC BU负责，AP部分驱动和OS，则由CRI负责。其架构图如下所示：

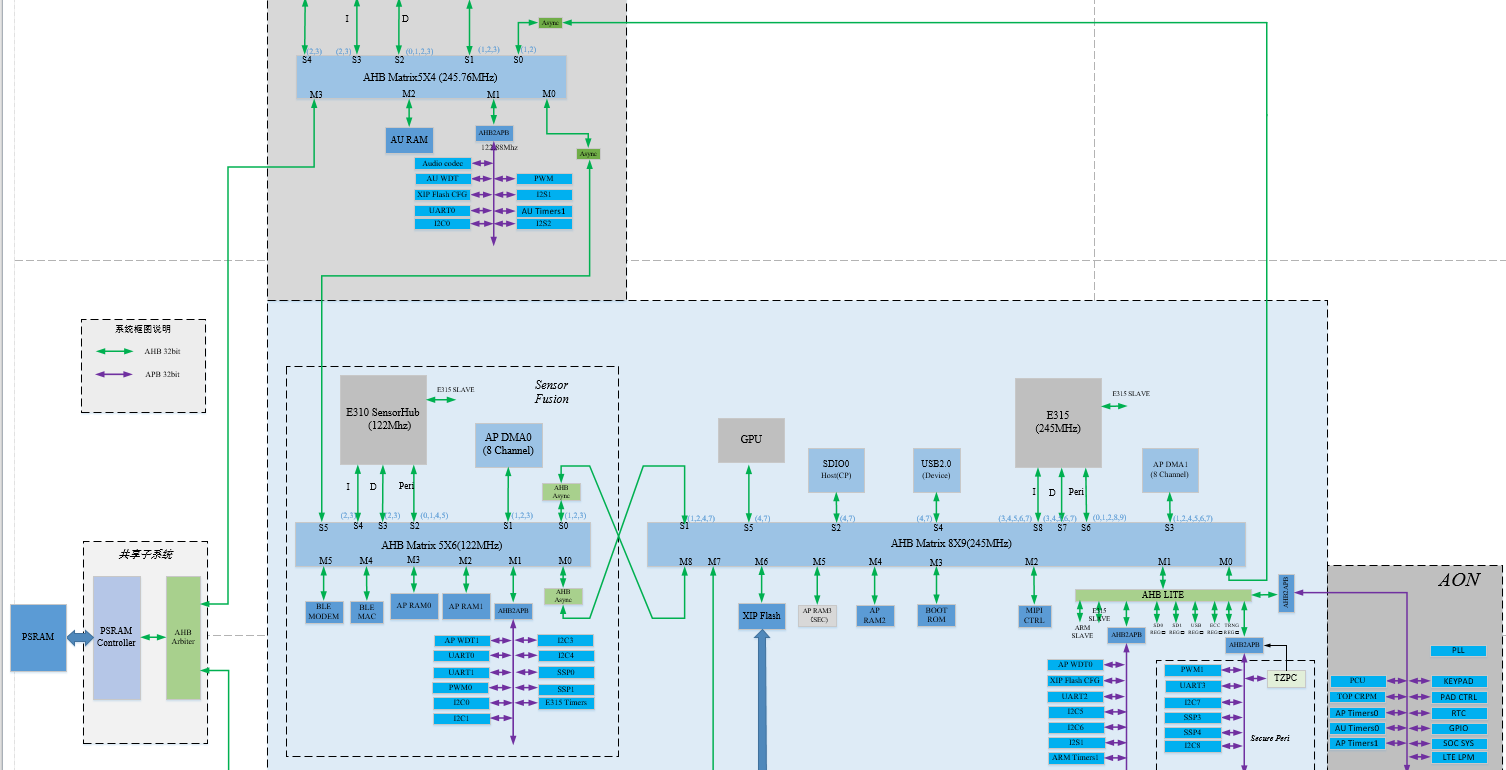


图1-1 轻智能手表AP部分SOC架构图

本文主要介绍AP及部分与CP交互的部分。其中AP采用E315（32bit），单核支持浮点运算，主频245M。CP交互部分为sensor hub（E315/ 122M ）和CAT（ZSP NANO），后续会仔细描述其交互接口的协议和方案。AP部分详细信息，如下：

* CPU模组：E315，245M，23K I/D TCM，16K I/D CACHE。其中TCM和CACHE容量可能会根据实际情况减小。
* 存储：8MB spiflash（XIP，支持片上运行），32MB RAM。
* BOOTROM：64KB，需要支持uart、spiflash、usb device、SD卡。
* 低速接口模块：GPIO、SSP、UART、I2C、WDT、TIMER、PWM
* 高速接口模块：usb device、sdio（接wifi模块或SD卡）
* 系统模块： DMA、PCU、CRPM（时钟管理模块）、PLL、RTC、
* 多媒体相关模块：I2S、2.5D GPU、mipi（controller）接触摸屏。
* 其它模块：OTP（存储training值，安全key，或客户其它用途）、ECC HASH PKA(安全启动 SC BU提供方案)

## 轻智能手表功能描述

目前市场反馈的手表功能支持较多，尤其是和手机交互和AI场景分析的需求较多。由于CRI目前主要工作还是集中在OS上，因此前期只会关注OS固有功能的实现需求分析上，其它部分暂不负责。

需要注意的是，CRI主要负责实现功能的demo，SC BU在此基础上进行二次开发。CRI负责功能如下图所示：



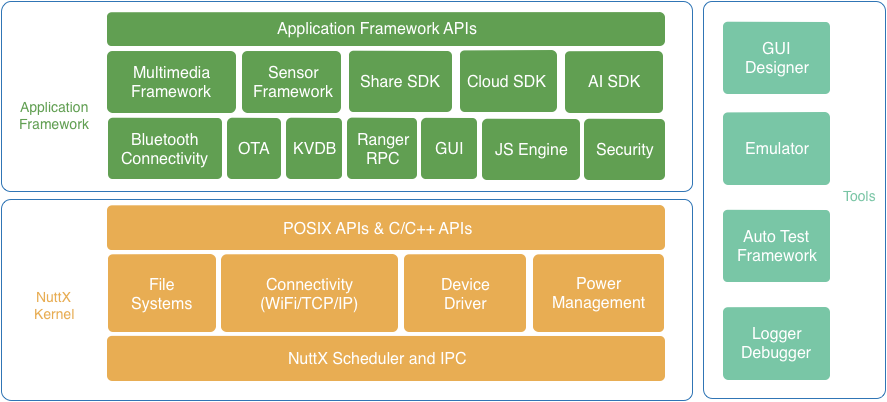
图1-2 轻智能手表CRI功能实现

轻智能手表的其它功能，建议主要还是SC BU负责。

# 主要开发工作内容概述

## 开发工作主要内容

面向IOT领域，即轻智能手表的OS包含模块很多，其大体架构图可参考小米的vela架构，如下所示：



因此开发工作可分为几个部分：

* 工具链

包括编译工具、emulator和IDE工具

* Bootrom开发

需要支持uart、spiflash、usb device和sd卡启动。

* 内核层

包括驱动的调试和移植、调度和IPC、网络协议（TCP/IP）、功耗管理、基础库、文件系统等。

* SDK

用于提供用户态的驱动接口，方便框架层的调用。

* 框架层

框架层的内容较多，包括编译框架、核心的JS引擎框架、用于显示和交互的GUI，还包括一些扩展的模块，例如：蓝牙、OTA升级、安全、多媒体/sensor框架等。

* 服务层

包括各种服务，例如音频管理服务、云服务、AI服务、电话管理服务等。

* 应用层

即轻型智能手表所需功能，详细见1.2节描述。

由此可见，OS牵涉到的组件和模块非常多，工作量也非常大。因此项目开发需要分阶段，分步骤地完成。

## 开发工作的规划

由于针对智能手表OS没有开源代码可供直接使用，只能自己参考其它OS并进行设计和实现。为了避免走太多弯路，因此需要对开发工作进行规划。这样不仅是后续工作的基础，更承担探索的重任；在具体实施上，将分为三个步骤逐步落实：

* 第一阶段：搭建js引擎和GUI （2022.1~ 2022.7）

主要目的是提供一个js应用程序开发环境，简单的实现PC上编译，单板上运行的目的。主要工作包括：编译框架、内核、文件系统、基础库（C/C++）、JS引擎和框架、GUI库。

* 第二阶段：OTA升级、蓝牙、multimedia/sensor framework、Cloud SDK（2022.8~2023.1）

主要目的实现升级、音频调试、蓝牙、sensor功能调试、modem连接实验。此时开发者除了sensor和电话是基本功能完成，其它模块大部分都已调试完毕。其中需要注意的是，音频功能要支持来电中断功能，sensor功能都已实现，性能还需调优。

* 第三阶段：电话和传感器全功能（2023.2~2023.7）

主要目的实现轻智能手表的全功能，此时准备正式版本的发布。

# toolchain工具链

toolchain主要工作包括：gcc（依赖基础库而不是glibc）、IDE和emulator的CPU部分等。其中工具IDE可参考树莓派工具，并实现界面app的开发和编译。要支持编译，可能需要先分析一下所依赖的库。这块工作可参考树莓派（arm）的工具，可提前熟悉和开展研究工作。

## gcc工具链

Gcc工具链在rv32imac(rv32gc)的ISA基础上，支持f单精度浮点指令和b位操作扩展指令，即支持rv32imafcb，其中b扩展指令对齐BitmanIP spec 1.0 rv32。包含对BitmanIP 1.0 gcc pass的部分优化，符号扩展优化，以提高位操作场景下编译器性能。

支持32位newlib的multilib特性，包括：rv32i、rv32iac、rv32im、rv32imac、rv32imacb、rv32imafc、rv32imafcb。

增加security属性扩展指令，编译和链接参数，增加security相关的attribute识别与处理，实现支持security特性的库。(待315项目落实)

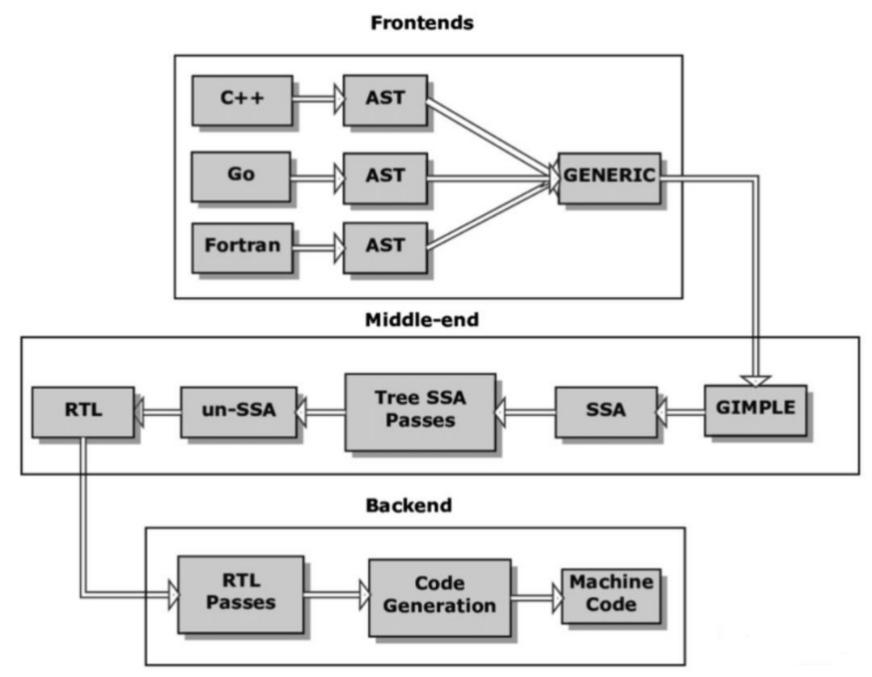


图3-1 GCC框架图

## IDE

### IDE整体功能与框架

IDE可以支持智能手表所选OS内核(例如NuttX）以及AI-SDK/ Cloud-SDK/Framework/JerryScript Engine等中间件代码的编辑、编译、加载、调试和运行，而对于顶层JavaScript应用是在引擎中解释执行的，因此也可以支持对JavaScript的直接编辑、加载、调试和运行，不需要预先编译好。

IDE前端采用的是基于Eclipse的开放平台，Eclipse 本身只是一个框架平台，除了 Eclipse 平台的运行时内核之外，其所有功能均位于不同的CDT插件中。开发人员既可通过 Eclipse 项目的不同插件来扩展平台功能，也可利用其他开发人员提供的插件。

整个IDE后台软件包可以认为是智能手表OS提供的全套SDK，由Code build tools、Toolchain、OpenOCD、Kernel（例如Nuttx）、Application Framework、JS Engine、图形化Qemu等软件包组成。其中GNU toolchain可类比为NDK开发包，包含GCC/GDB/Newlib/ Multilib/Security/Sysroot等等模块组成，支持RV32 imafcb的ISA。

整体IDE框架如下图：

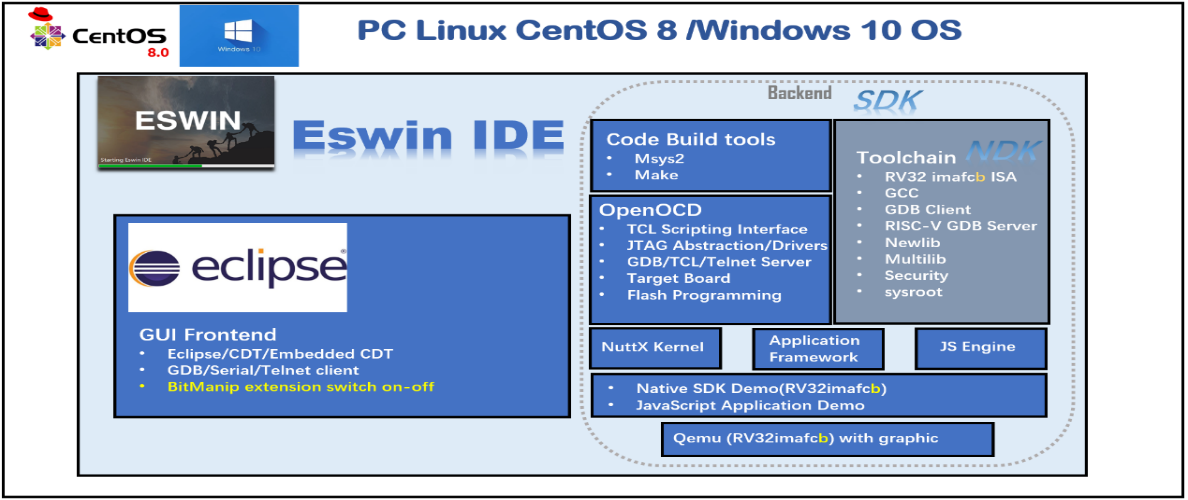


图3-2 IDE架构图

### Native程序的调试

Smart Watch项目的Native程序可分为两类：

1. ELF嵌入式C/C++程序，如NuttX操作系统；
2. 基于NuttX操作系统的其他程序，如JerryScript Engine, Cloud-SDK, AI- SDK, Framework API等Framework Application中间件程序。

这两类程序可以继承和参照已有的Eswin IDE对elf程序如OpenSbi/U-boot/Linux Kernel 和Guest Linux OS App的调试方法。采用IDE前端调用Gdb+OpenOCD或者Gdb+Qemu的方式可以对NuttX Kernel进行调试；采用Remote Gdb+硬件或者Remote Gdb+Qemu的方式可以对Nuttx操作系统基础上运行的native程序进行调试。具体调试手段包括设置软硬件断点、flash断点、watchpoint功能，step/step into、next/next into、continue等，变量值、表达式、内存内容显示，C/C++高级语言和反汇编单步视图显示等，如下图所示：

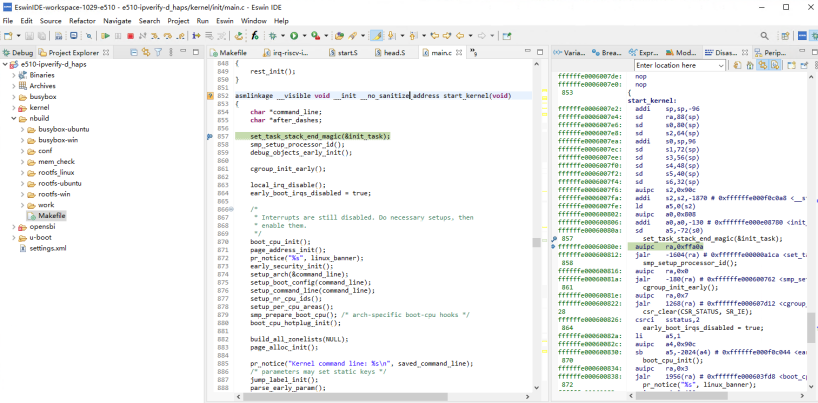
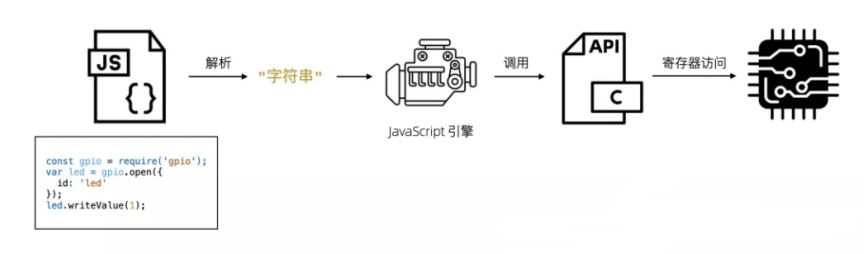


图3-3 Native程序调试

### JavaScript应用程序的调试



目前APP主要是由运行在JerryScript引擎基础上的JavaScript开发的，因此面向APP的可视化IDE需要完成JavaScript的编辑/开发和调试，使之可以在搭载操作系统和JerryScript引擎的目标硬件或模拟器Qemu上正常运行。 JerryScript 提供了一个远程调试器，允许调试 JavaScript 程序。调试器有两个主要组件：作为 JerryScript 二进制文件一部分的服务器和单独的客户端应用程序。当前，在JerryScript的/jerry-debugger 子目录中提供了基于 Python 的调试器客户端。

1. 设置调试器服务器

以下参数必须传递给tools/build.py：

--jerry-debugger=on

1. 调试JavaScript

在 JavaScript 应用程序运行之前，调试器客户端必须连接到服务器。引擎初始化后立即支持多个文件的动态附件（此功能可用于 python 客户端）。JerryScript 不保留调试信息（例如，每个可能的断点位置的行索引）。客户端应在具有更多资源的系统上运行，并且应该能够存储此信息。JerryScript 在传输到客户端后释放所有调试信息以节省内存。

以下参数使 JerryScript 等待客户端连接：

--start-debug-server

以下参数使 JerryScript 等待客户端源代码：

--debugger-wait-source

还建议提高日志级别以查看等待客户端连接消息：

--log-level 2

Python 客户端可以通过在命令行上指定其 IP 地址来连接到服务器。如果服务器和客户端在同一台机器上运行，则地址可以是 localhost。

建立连接后，调试器可以控制执行。调试器总是在第一个可能的断点位置停止。效果与使用stop命令相同。这允许在执行的有意义部分开始之前插入断点。

可以通过该help命令查询客户端所有可用的命令。

Eclipse本身具有应用于web开发的JavaScript编辑开发调试插件**Eclipse JavaScript Development Tools**，因此对于IDE前端界面比如Debug视图显示可以借鉴其功能，对于IDE后台的debug工具则需要调用JerryScript自带的python调试器，实现与目标上运行程序的通信，因此需要增加Debug的配置界面。JavaScript在eclipse中的Debug试图界面参考如下：

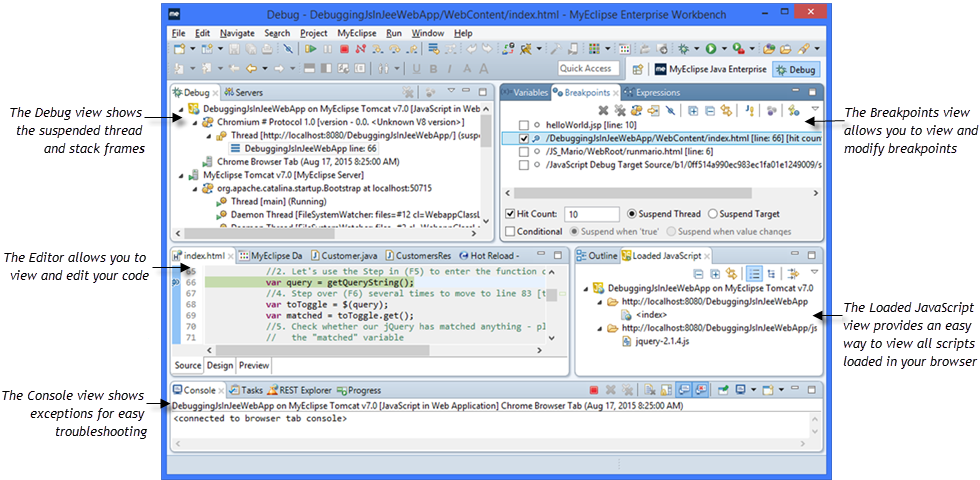


图3-4 JavaScript调试示意图

由此可见，我们基于我们已有的Eclipse编辑调试环境，理论上也可以实现 JaveScript的调试和运行。

# emulator

emulator包含两部分：CPU部分和SOC部分，CPU部分由toolchain提供，SOC部分由应用开发部提供。

与硬件环境和gcc工具链匹配，选取Qemu V6.0为基础版本，模拟支持BitmanIP spec 1.0扩展指令的硬件单元，即支持ISA为rv32 imafcb的指令集架构。支持安全属性，模拟安全指令硬件单元，与gcc工具链、操作系统的安全属性相匹配。

使用Qemu的system模式，增加模拟CPU型号e315的machine设备，包括模拟的CLIC中断，32M内存RAM，icache/dcache, ITIM/DTIM，BUS等CPU最小系统，以及8M SPI FLASH存储单元，低速接口模块GPIO、SPI、UART、I2C、WDG、TIMER、SDIO，高速接口模块USB、BLE等等智能手表项目所需的各类SOC外设。

在基础硬件模拟完善的基础上，Qemu可以支持智能手表所选OS内核（例如NuttX Kernel）的加载运行，以及基于此内核中间件程序（如AI-SDK、Cloud-SDK、Framework、JerryScript Engine等等）的加载运行，最上层支持JavaScript应用软件的执行。

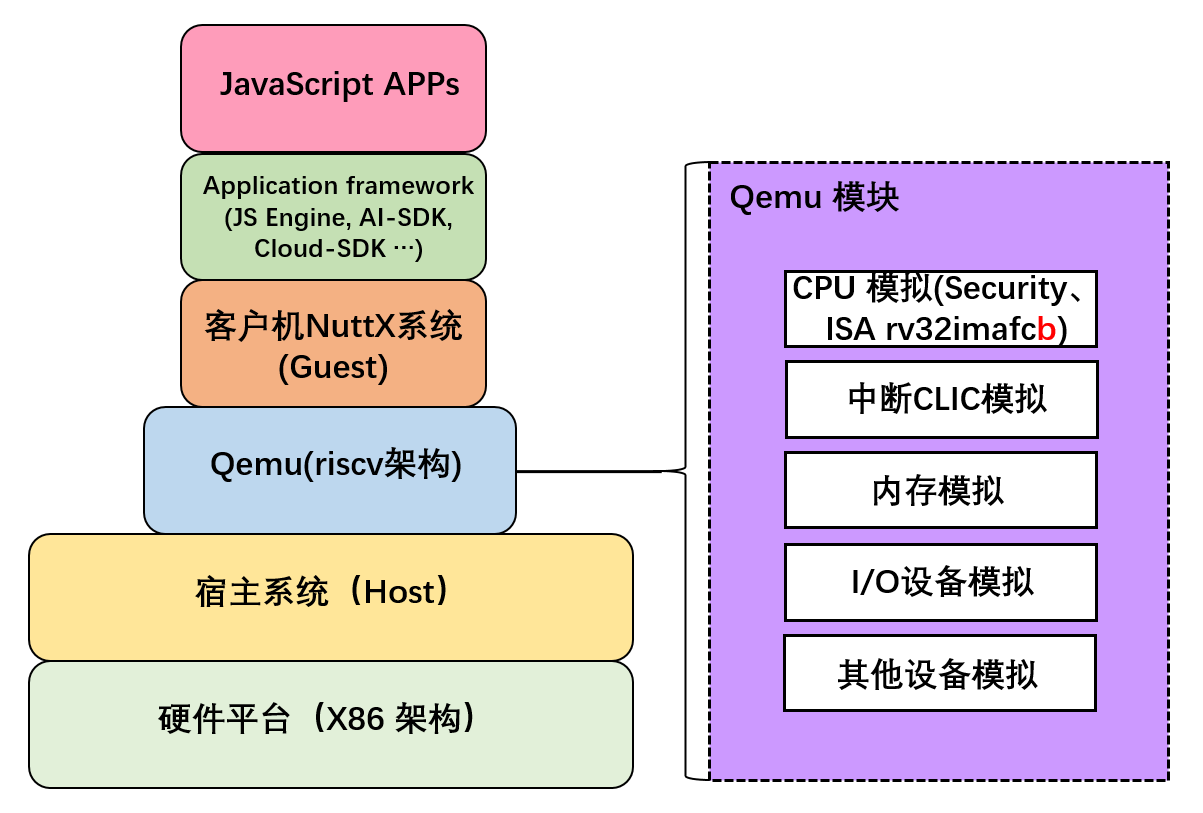


图4-1 RISC-V Qemu架构图

支持图形化界面，类似Android Emulator，支持应用程序可视化操作和显示。一方面需要模拟器支持界面规格等配置，另一方面需要OS内核支持图形绘画和渲染。类似下图界面：



图4-2 Qemu图形界面示例图

# 内核层

内核选用posix接口兼容性较好的Nuttx，这样可以降低集成第三方库的难度。Nuttx提供最基本的任务调度、跨进程间通信、文件系统等基础OS功能，同时也提供简洁高效的设备驱动、轻量级的TCP/IP协议栈和电源管理、CP对接模块等组件。

CP对接模块比较特殊，另起一章进行描述。本章主要描述AP侧工作，以下分几个方面来描述。

## 设备驱动

设备驱动主要分为两部分，AP部分驱动和对接CP模块的驱动。AP部分主要模块见1.1节，这边主要从子系统的角度进行描述：

* 显示子系统：

主要包括触摸屏spi接口、2.5D-GPU、mipi

* 音频子系统：

主要支持本地播放、在线播放、电话等音频相关部分。负责的模块有：I2S、audio codec（sensor hub侧）

* 连接子系统：

主要包括USB device、sdio。

* 功耗管理子系统：

主要包括pll和门控时钟，如果SOC提供电源分区，则可以进一步降低各个模块的功耗。负责的模块有：PCU、CRPM、PLL

* 时钟子系统：

主要包括WDT、TIMER、RTC等。

* Sensor hub子系统

主要由SC BU负责，与AP的接口部分，需SC BU提供方案和协议，以及测试等资料。需要注意的是，蓝牙也属于sensor hub子系统负责。

* 电话子系统（基带CAT）

主要由SC BU负责，与AP的接口部分，需SC BU提供方案和协议，以及测试等资料。

其它模块：GPIO、SSP、UART、I2C、PWM、DMA

以上要注意的是：CP和AP的连接系统方案目前还没定，暂定为数据通过共享内存，使用查询的方式更新数据。打开app后，直接查询数据。

## 协议栈

其主要模块包括蓝牙协议栈、TCP/IP协议栈等。

## 功耗管理

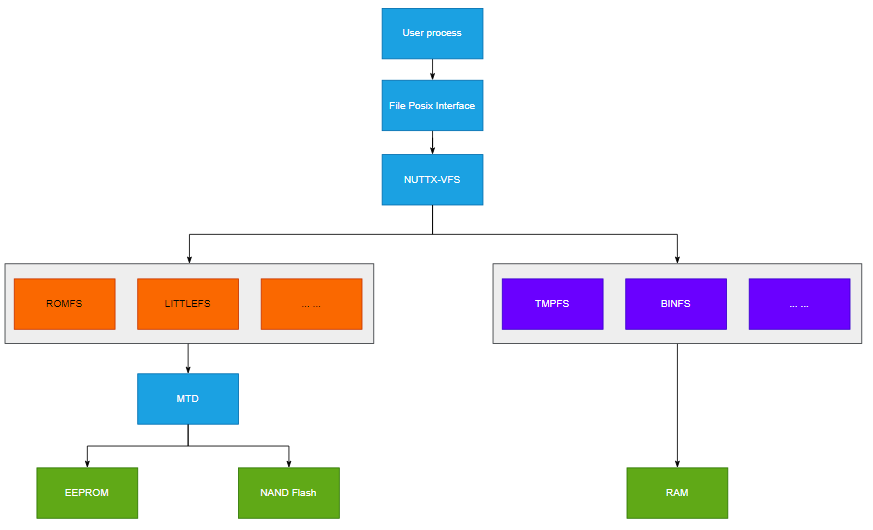
由于手表对功耗十分敏感，因此功耗管理模块十分重要。主要根据SOC的门控时钟或电源分区等相关低功耗设计，提供相关api函数接口。

功耗管理方案需SOC提供。

## 文件系统

文件系统，是对一个存储设备上的数据和元数据进行组织的机制，它是操作系统管理持久性数据的子系统，提供数据存储和访问功能。

将一个文件系统与一个存储设备关联起来的过程叫做挂载(mount)，挂载时会将一个文件系统附着到当前文件系统层次结构中(根)，在执行挂载时，需要提供文件系统类型、文件系统和一个挂载点。



上图所示为用户程序调用存储设备操作的文件系统架构关系图。以下分别描述这几个层次。

 用户层，通过POSIX接口调到VFS层的通用接口；

 VFS层，相当于一个适配层，用于对接不同的实际文件系统；

 实际文件系统层，典型的情况下一个文件系统都需要绑定到块设备驱动程序上，而一些不太典型的情况是不需要块设备驱动。在Nuttx中，需要块设备驱动的文件系统为：ROMFS、LITTEFS等；而不需要块设备驱动的文件系统为： BINFS、TMPFS等；

 MTD，Memory Technology Devices，向上提供MTD接口，向下对接不同的硬件设备；

### Nuttx VFS

NuttX 包括一个可选的、可扩展的文件系统。这个文件系统可以完全省略；NuttX不依赖于任何文件系统的存在。

1. 伪根文件系统。默认情况下可以启用一个简单的内存中的伪文件系统。这是一个内存文件系统，因为它不需要任何存储介质或块驱动程序支持。相反，文件系统内容是通过标准文件系统操作（open、close、read、write等）引用的即时生成的。从这个意义上说，文件系统就是伪文件系统（类似于Linux /proc文件系统）。

任何用户提供的数据或逻辑都可以通过伪文件系统访问。通常把伪文件系统目录中的字符和块驱动程序放置在/dev中。

1. 挂载的文件系统。简单的内存文件系统可以扩展挂载块设备，提供对通过一些大容量存储设备备份的真实文件系统的访问。NuttX 支持标准mount()命令，该命令允许将块驱动程序绑定到伪文件系统中的挂载点上。
2. 与 Linux 的比较。从编程的角度来看，NuttX 文件系统看起来与 Linux 文件系统非常相似。但是，有一个根本的区别：NuttX 根文件系统是一个伪文件系统，真正的文件系统可能挂载在伪文件系统中。相比之下，在典型的 Linux 安装中，Linux 根文件系统是真正的文件系统，而伪文件系统可能会挂载在真正的根文件系统中。NuttX 选择的方法旨在支持从非常小的平台到中等平台的更大可扩展性。

NuttX支持多种文件系统，例如tmpfs、littlefs、romfs、procfs等。以下简单介绍下littlefs和romfs。

### Littlefs文件系统

由于系统资源有限，因此需要选择一种尺寸较小的文件系统。Littlefs是ARM公司开源的文件系统项目，专门用于解决spi flash在嵌入式系统上的文件管理，是一种转用的嵌入式文件系统。主要有三个特点：

1. 掉电恢复

在写入时即使复位或掉电也可以恢复到上一个正确的状态。

1. 擦写均衡

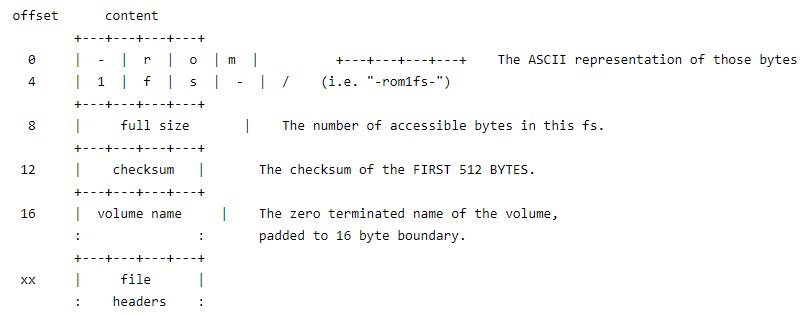
有效延长flash的使用寿命。

1. 节省ROM/RAM空间

除此之外，还需提供一层VFS虚拟文件系统，为各种文件提供统一的操作接口。它是具体设备文件和文件系统之上的抽象层。其目的是允许应用程序以统一的方式，访问不同类型的具体文件和设备。用户可以将ramfs，littlefs等具体的文件系统注册到VFS中，然后使用标准操作接口访问其中的文件。

### Romfs文件系统

Romfs是一种简单的只读文件系统，因为它的小巧性和内存结构简单，一般常用来系统启动初始化的文件系统。romfs基本格式定义在内核源代码树下的“Documentation/filesystems/romfs.txt(.rst)”。同时可以使用命令genromfs来制作romfs文件。如下为romfs数据格式：



如下为在romfs文件系统下，加载和运行的elf文件模型：



图5-1 加载和运行elf文件模型

## 基础库

由于嵌入式系统，对空间要求十分苛刻，因此绝大部分IOT的OS都不使用glibc，甚至连android也是google自己开发的一套基础库（bionic），以便减少尺寸。系统前期采用nuttx自带的基础库，后期可根据需要，替换更为精简的基础库。

## 调度系统

内核中的调度往往在一些特定的调度点完成，比如信号量的使用及释放、消息队列中消息的接受发送、信号的传递等等，而nuttx提供了完整的POSIX API，比如pthread、信号量、消息队列、时钟/定时器、信号、环境变量等，并支持大多数RTOS都没有实现的进程概念。由于一些机制的数据通过inode节点存储，且接口通过文件系统相关接口进行操作，所以与文件系统的关联较大，比如消息队列与信号量；而像信号的实现机制，则与文件系统无太大关联。



## SDK

SDK包含一系列文件的组合，包括库、源文件、文档、示例等等。其提供的用户编程接口和功能模块为其主要功能，emsis的SDK主要分为处理器、SOC、板级、外设、HAL层、应用demo等几部分，下面分别介绍主要部分。

### 处理器部分

处理器部分除了RISC-V的标准部分，还有其扩展部分。因此SDK的处理器部分提供了CORE API、DSP和NN（神经网络）三大组成。其结构图如下所示：

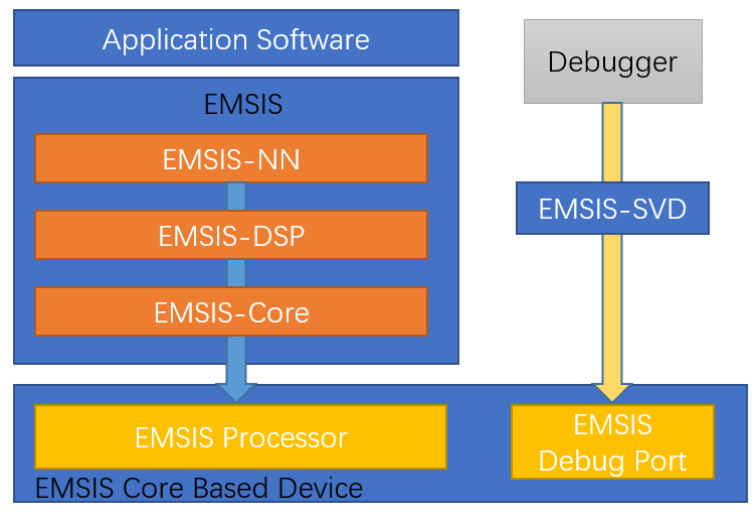


图5-1 CORE API主要架构图

由上结构图可知，Eswin的SDK主要提供core、dsp、nn三大块，其特性如下：

* Eswin Core api ：eswin处理器和外设提供的标准api函数接口。
* Eswin DSP：eswin支持的定点、单精度浮点、以及用SIMD指令优化的DSP库。
* Eswin NN：eswin提供优化的最优性能，最小内存损耗的神经网络功能。

由于DSP和NN属于SDK的扩展部分，产品可根据实际需求动态配置。CORE API属于处理器的重要组成部分，其又可以分为以下几个层次。

* 硬件抽象层：提供CSR寄存器、定时器、CLIC、PMP寄存器和CPU的访问接口。
* 标准的中断和异常访问接口。
* 系统初始化接口函数。
* intrinsic指令函数接口。
* 系统时钟函数接口。

通过CORE API提供的函数接口，我们可以访问CPU的基础能力。

### SOC和板级部分

SOC和板级部分主要用于最小SOC系统，包括cpu、sram、总线、DDR和基础外设（UART）等。这块根据产品具体配置，会有较大差异。

### 外设部分

由于每个SOC芯片，外设部分都会有差异，这块由芯片厂家提供。Eswin SDK提供的外设SDK也分为两部分，SOC芯片外设和平台外设部分。

### HAL

Eswin的硬件抽象层提供主要以下接口api：base、HPM、PMP、LOCK、TIMER、memory、TICK、中断和UART。

# 编译框架

本编译框架参考鸿蒙的框架进行搭建。系统功能按照“系统 > 子系统 > 组件”逐级展开，支持根据实际需求裁剪某些非必要的子系统或组件。

本编译框架主要提供以下两个功能：

1. 构建某一开发平台（E315）的产品（平台指的是开发板和OS的组合）；
2. 根据产品配置按照组件组装打包产品需要的能力（组件（components）是对子系统(subsystems)的进一步拆分，即一些可复用的软件单元，包含源码、配置文件、资源文件和编译脚本，可以独立的构建）。

## 编译架构与构建

编译构建流程主要包括编译命令行解析，调用gn以及执行ninja。

1. 命令行解析：解析待编译的产品名称，加载相关配置；
2. 调用gn: 根据命令行解析的产品名称和编译类型，配置编译工具链和全局的编译选项；
3. 执行ninja：启动编译并生成对应的产品版本。

编译构建流程图如下所示（图6-1）：

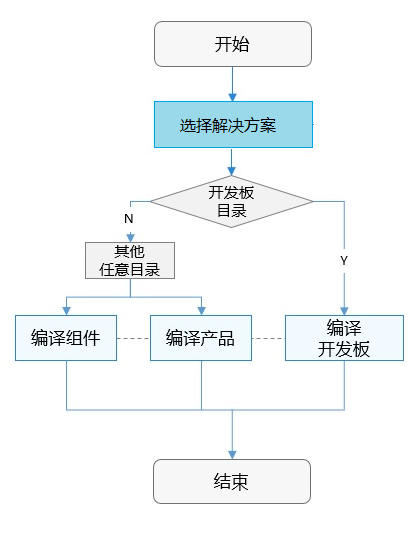


图6-1 编译构建流程图

1. 配置选项——选择解决方案：
   * + 确认开发平台配置文件（.开发平台下json文件）；
     + 配置工具链（工具链toolchain目录下的.json文件）；
     + 配置子系统及组件（检查各子系统下的依赖内容，生成完整的组件树）。
2. 编译构建：
   1. 使用gn gen命令，根据.gn、.gni文件规则读取产品配置(主要包括开发板、内核、选择的组件等)生成解决方案out目录和ninja文件。即根据生成的组件树，生成各组件的构建脚本。
   2. 调用ninja\_build，启动编译依次编译输出各自的中间文件，最终生成可执行文件。此过程生成.o文件、.so文件、.bin文件等格式文件。

具体过程：编译源码（生成一个 a.out），并在当前目录生成一个 ninja.build（这个文件类似于 make 工具的 Makefile，语法和规则非常类似）。根据ninja.build 重新编译生成可执行文件 ninja，在 ninja 根据 ninja.build进行编译时自动创建一个 build 目录用于存放编译过程中的临时文件，比如 \*.o 。

* 1. 镜像及打包：将组件编译产物打包，制作文件系统镜像。

1. 清理文件：

执行ninja clean，清除out目录对应产品的编译产物。清除指定路径可输入路径参数：ninja clean out/board/product，默认将清除当前配置的产品对应输出路径。

## 文件系统与flash分区

编译代码后，需将生成的文件打包成对应的文件系统，然后烧写到移动终端。

一个系统可以存在多个不同的文件系统，文件系统存于闪存(flash) 、硬盘(mmc)等介质上。功能要求可对flash进行分区，每个分区可被格式化成不同的文件系统。各分区选择相应合适配套的文件系统。

文件系统制作包括两部分：制作镜像文件，对生成文件进行打包。

### 文件系统生成

此处以jffs2根文件系统（日志文件系统）为例：

1. 编译整个目标文件：

* 逐个遍历子目录中的.ninja文件；
* 生成\*.map \*.bin \*.asm等文件。

1. 编译apps 目录下的各个APP；
2. 创建根文件系统下的各个目录( /bin 、 /app 、/lib... )；
3. 使用 prepare 创建/musl 目录，并将 c/c++ 库拷贝到该目录下；
4. 生成镜像文件 rootfs\_jffs2.img ，调用 mkfs.jffs2 来制作 jffs2 文件格式的镜像；
5. 最后用 zip 命令将 rootfs 打包成 rootfs.zip。

以上为根文件系统的制作过程。计划将增加了一个/ out 目录，其中rootfs 便为制作的根文件系统；rootfs\_jffs2.img 为镜像文件，可以烧到flash 中。

### Flash分区

智能手表项目的功能要求中涉及OTA升级等。基于此，为保证数据安全，需要提前合理分配flash，实现当前运行的程序、升级数据、系统参数等数据的存储。下图（图6-2）为flash分区设计图，各分区的起始地址和“区”的大小根据实际开发需求进行设置。其中，预留区作为未分配存储区，可根据后期实际功能需求进行划分。

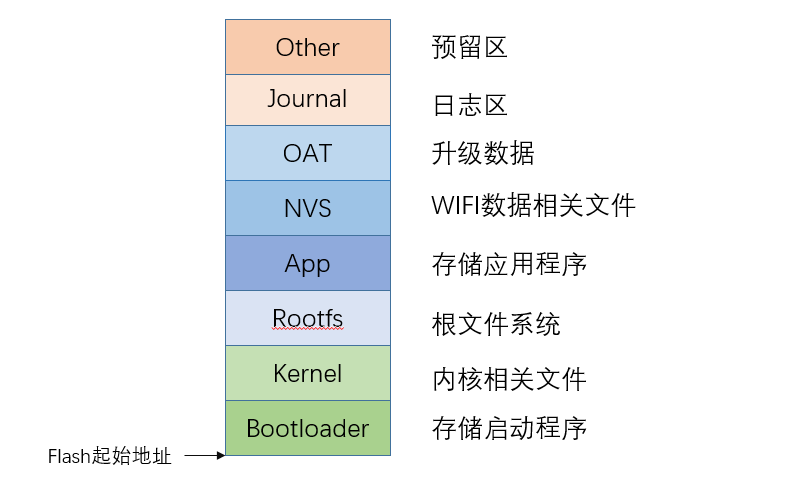


图6-2 flash分区设计图

# JS引擎和框架

Js引擎选择jerryscript，此引擎的性能直接关系到手表应用程序的运行效率。且还需提供底层驱动的一起接口和框架，因此十分重要。

## JS引擎概述及原理

我们选用Jerryscript来作为JS引擎，并用一个demo来解释其工作原理，如下图：

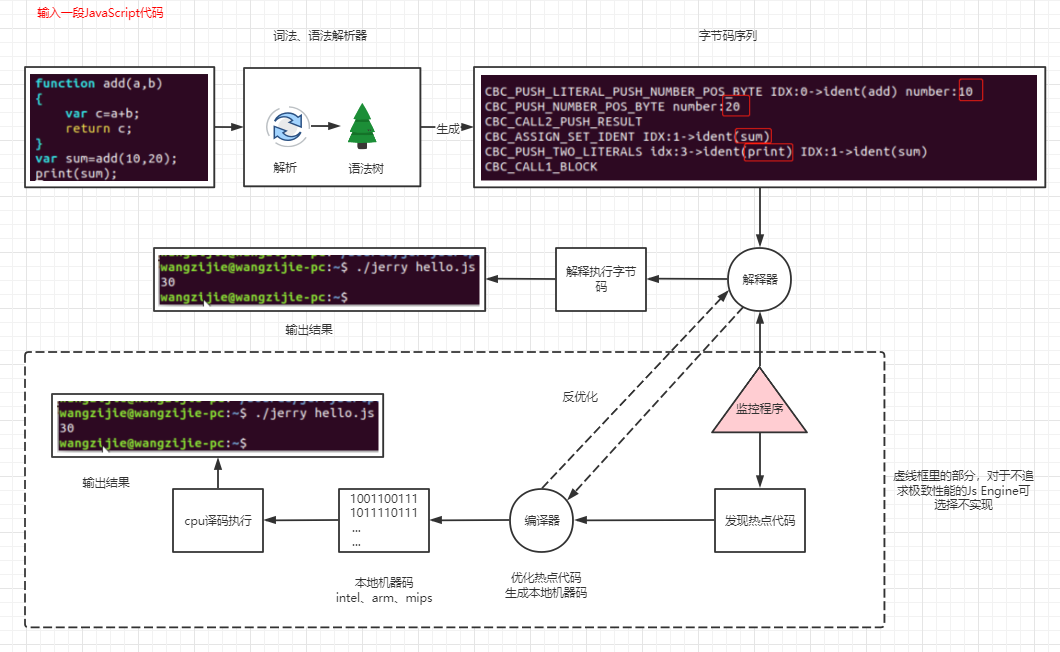


图7-1 jerryscript的基本工作原理

上图所示代码功能是调用add函数，完成10和20相加，并打印结果。以下则是JS引擎执行流程：

1. 编译完JerryScript可以得到一个叫jerry的ELF文件，这个就是JS引擎。
2. 接着输入一段JavaScript代码至hello.js，并运行./jerry hello.js。
3. 此时引擎本身会对hello.js做词法、语法解析生成抽象语法树。
4. 经过语义分析生成字节码序列，可以看到变量及符号能够在字节码序列中体现。
5. 然后引擎的解释器会解释执行生成的字节码序列，最后执行输出结果30。

有的JS引擎为了追求极致性能，采用以空间换时间的策略。会选择上图虚线框中的部分执行（虚线框的部分引擎可选择实现）。执行流程还有会如下：

1. 热点监控程序，对解释执行的代码进行监控追踪，当发现被监控代码执行到一定次数后，会直接将中间代码交给编译器处理。
2. 编译器会直接将优化后的中间代码直接编译成本地机器代码，CPU直接执行机器代码输出结果30。
3. 此外如果热点代码部分有改动，监控程序会及时发现并执行反优化操作，使用解释器再去解释执行，不过此过程会有些性能开销。所以引擎中编译器的设计也是个值得探讨的话题。

## JS框架分层设计

之所以除了JS引擎之外，还需设计JS框架并进行分层，主要有以下两点原因：

1. 大多数JS引擎支持功能较少，如部分硬件驱动、文件系统、网络、GUI等都不支持。
2. 设计JS应用框架，并以此做JS引擎的适配工作，可以实现对引擎功能选择性扩展。

基于以上目的，我们设计了如下架构图：

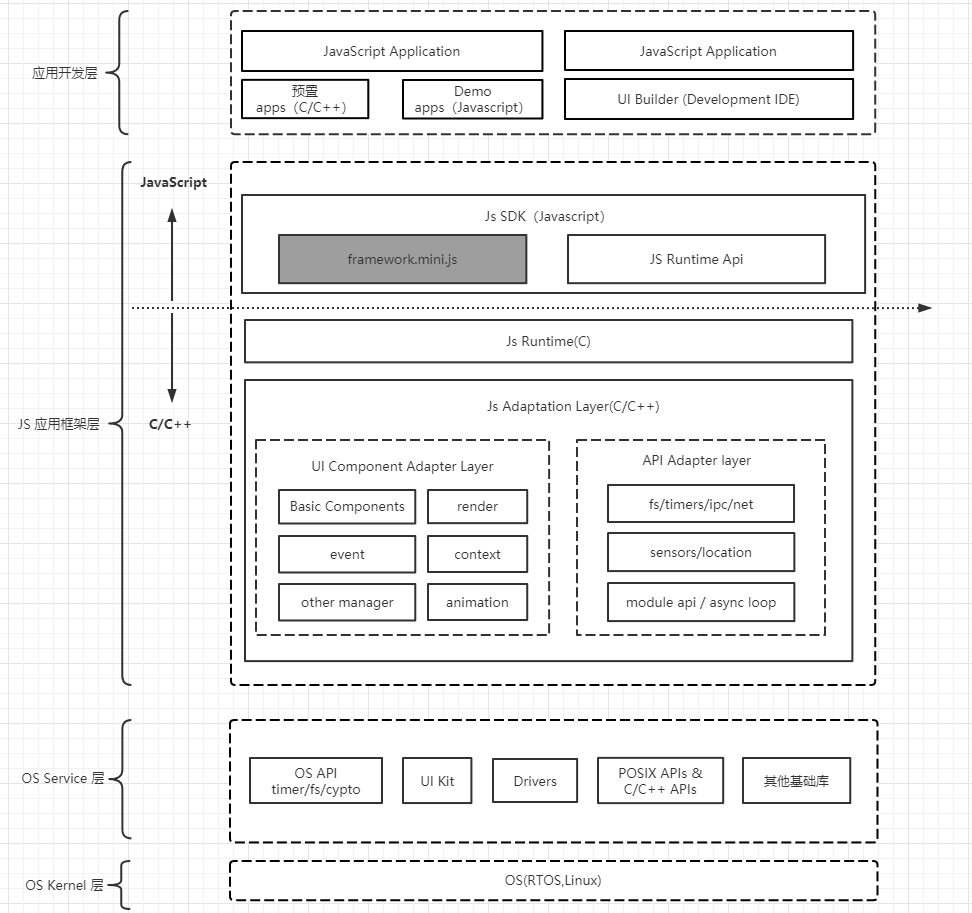


图7-2 JS引擎框架的分层设计

由上图可知，对于整个手表系统站在JS框架的角度可以分为四层，从下到上依次是OS kernel层、OS Service层、JS应用框架层和应用开发层。其中工作主要集中在OS Service层和JS应用框架层。其中OS Service层主要是对内核用户态进行封装，提供核心API和SDK。JS应用框架层是我们分析的重点，同时也是JS框架工作量最大的地方。下面依次分析JS框架各个层级。

### JS Adaptation层

设计JS应用层框架时，需要兼容下层内核及硬件平台。由于下层的多样性，因此需要一层抽象层，便于向上提供统一的接口。JS Adaptation就是这个抽象层，可以对接不同的内核及硬件平台，并提供上层所需的系统服务。这样对于下层的改动，我们只需要修改JS Adaptation层即可，不再影响到上层次的逻辑。

适配层按照提供的功能是否可视化，来分为UI Component Adapter 层和API Adapter 层，主要使用C/C++语言实现，下面两节进行详细说明。

#### UI Component Adapter 层

由于UI Kit的选用是不确定的，这也不是JS 应用框架的主要工作。为了适配系统服务，提供的不同的UI Kit，我们设计了UI Component Adaptation Layer。主要用来和不同UI Kit进行适配工作。下面列出了需要设计的主要功能模块。

* Basic Components 提供基础的图形组件适配能力。
* Render 提供图形组件渲染适配能力。
* Evnet 提供图形组件上的事件机制适配能力。
* Context 提供图形组件上文机制适配能力。
* Animation 提供图形动画适配能力。
* Other Manager 提供其他还未列出的图形组件管理适配能力。

#### API Adapter 层

对于系统服务的实现，不同的方案选择实现的方式也不一样。为了解决不同系统服务的实现方式，支持灵活配置。我们设计了一个API Adapter Layer，主要用来和不同平台的系统服务之间做适配工作。下面列出了需要设计的主要功能模块。

* Basic API：主要是系统基础库提供的API功能。一般而言嵌入式设备为了减小系统服务的体积，会提供自己专有的基础库。其中一些功能接口遵从POSIX标准，其他的一些可能因为平台不同，实现也大不相同。为了彻底解决接口不兼容的问题，需要设计适配接口，完成相关的适配功能。
* Drivers：不同平台的系统服务实现与使用方式不同，因此需要做适配功能。
* 架构及OS适配：不同的架构和OS平台，需要做适当的适配工作。如:1、可能涉及到的硬件指令集相关的问题。2、OS中启动方式的配置及启动代码的编写等。

### JS Runtime层

JavaScript代码的运行，需要可靠的运行环境，这些运行环境构成了Runtime层。Runtime层主要由JS引擎适配模块、JS引擎和众多内置功能模块构成。

#### JS 引擎适配层

JS框架可能会选择不同的JS引擎，另外大多数的嵌入领域的JS引擎所提供的功能都比较少。因此大多都提供可扩展功能，让开发者结合使用场景，选择一些内置功能添加到引擎中去。对于JS引擎本身和内置功能，它们应该是可灵活配置的。分层的思想就是要在它们之上再加一层适配层，来分别适配不同的引擎以及相关的内置功能。因此理论上JS引擎适配层应当在它们两者之上。

#### JS引擎

我们选择了JerryScript作为引擎层的实现，后面如果有更合适的JS引擎，则可以修改适配层再次匹配。目前选择JerryScript作为JS引擎，主要基于以下几点考虑：

1. 有可参考的实现，业内比较成熟的产品，华为手表系统架构中，JS引擎选取的就是JerryScript。
2. 相比于其他的开源JS引擎来说JerryScript足够小，比较适合手表这样的嵌入式设备。
3. 架构相对其它JS引擎来说比较简单，源码也比较少，学习和研究周期会比较短。另外由于其提供了模块注册功能，比较适合做扩展集成。

### JS 引擎内置功能

对于内置功能，下面列出比较重要的一些需要在JS应用框架中完成的内置功能。这些功能根据JS引擎不同的需要，可能要做适配程度也不同。

|  |  |  |
| --- | --- | --- |
| 编号 | 功能 | 描述 |
| 1 | Buffer缓冲区 | 为javaScipt提供使用Buffer缓冲区的能力 |
| 2 | 控制台日志 | 完成打印日志、记录错误、警告日志信息等，便于调试。JS引擎需要log能力 |
| 3 | 软件加解密 | 应用层软件可能需要安全验证，JS引擎需要提供加解密功能 |
| 4 | 文件操作 | 应用层软件需要操作文件系统的能力 |
| 5 | 进程间通信 | 应用层软件需要进程间通信能力 |
| 6 | IO操作 | 应用层软件有的可能需要进行IO读写，需提供相关功能。 |
| 7 | 本地共享库加载 | 应用层软件可能因性能有要求，需要本地代码实现，需提供加载本地共享库能力。 |
| 8 | tcp/udp | 应用层软件需要与服务端建立tcp/udp连接传输数据。 |
| 9 | http | 应用软件需要请求http服务器获取超文本或其它文本数据。 |
| 10 | tls | 安全传输层协议，为网络通信提供加密访问手段。JS引擎需提供基础的tls能力为http及其它通信手段做加密支持。 |
| 11 | mqtt | 物联网设备和云平台之间的通信手段。 |
| 12 | websocket | 为了构建完整的网络通信手段，websocket也需要提供支持。 |
| 13 | 软件定时器 | 满足系统的软件定时需求。 |
| 14 | JS事件管理 | 需要提供事件管理能力，满足应用开发过程中对事件的使用。 |
| 15 | JS模块管理 | 为了实现OS的灵活性，需提供模块化管理能力。 |
| 16 | 系统服务 | 对于电话、短信等应用中所需的服务，应用需要授权读取相关信息。 |

当然，应用需求多种多样，所需内置功能也不仅仅只限制于以上功能，因此后面还需根据需求，开发所需功能模块。

### JS SDK层

framework.mini.js及 JS Runtime API共同组成JS SDK平台层，此层主要给开发者提供API支持。

#### Framework.mini.js（可选实现）

这个功能，相对于整个JS应用框架来说是可选实现项，也就是说可以不实现这一模块。因为它的功能只基于JS Runtime API为用户对常用功能提供进一步的封装，即为用户提供一些JS应用层框架设计人员已经实现的一些API。

#### JS Runtime API

从实现上来说JS Runtime API应该是在Runtime层实现的。但其API却是暴露在JavaScript代码中使用的。所以我们把这些Runtime提供的API也归到JS SDK层中来。

### 应用开发层

应用开发层的功能已经超出了我们对JS应用框架设计的范围，因为大部分应用程序都是由用户去开发的。但此层工作量还是蛮大的，总体来说主要需要完成如下功能：

1. 系统预置apps，这些app是系统自带，其主要是C/C++语言实现，由操作系统提供实现。
2. Demo apps，系统提供的示例app，使用JS语言开发，主要提供给用户参考开发，额外提供不单独内置于系统中，由JS应用框架设计者选择实现。
3. 3rd part apps，主要是第三方的app，由应用层软件开发者（主要是Javascript程序员）实现。
4. UI Builder：对象JS SDK提供的组件化开发还是不够方便，更多的Javascript开发者可能希望可以提供一个可视化的组件平台，且可调试的IDE。基于这个IDE进行应用层的软件开发。

# 显示子系统

手表的显示子系统包括2.5D GPU、mipi dsi和触摸屏，显示参数指标：500\*500\*32BPP@60。GPU的IP采用Verisilicon公司的GCNanoUItraV，mipi dsi采用synopsys公司的IP。两者之间采用DDRless技术直通，即GPU和后者之间不需要DDR提供共享内存，Framebuffer直接集成至GPU内部。 由于GPU和mipi dsi直通，因此需要考虑两者的连接方式和数据格式的兼容，查阅资料大致可以推断，GPU数据格式采用yuv compression，mipi dsi采用DPI接口（支持yuv数据格式）。具体直通方案，由SOC硬件设计人员进行确认。

手表的显示屏还需支持熄屏显示，熄屏时通过spi接口显示时间。因此显示子系统方案还需考虑spi接口的送显。其大致流程框架如下所示：

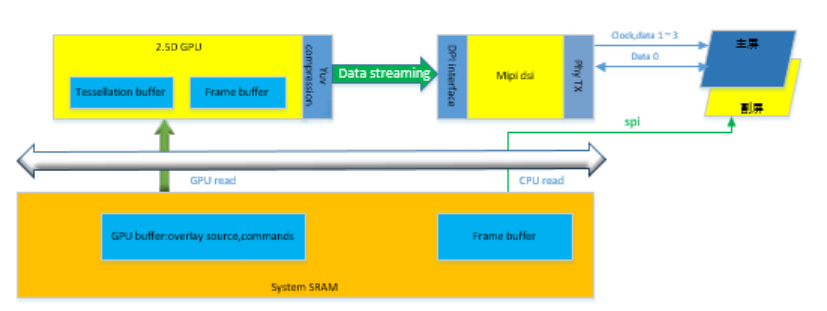


图8-1 显示子系统框架图

从上图可知，显示子系统有两条路径进行显示，因此工作也分为两块。主屏通过GPU直通mipi进行显示，因此需要移植gui库支持2D硬件加速。副屏用于显示时间，画图工作量较少，可通过cpu（freetype）进行画图，然后通过spi送显。

需要注意的是，freetype需要的字库比较大，可以采用外部扩展的存储设备(emmc)保存字库。

## GUI库

支持嵌入式的gui有很多种，整理了一下常用的嵌入式gui库并进行对比，对比列表如下所示：



经过比对，选用占用资源较少的lvgl作为手表的gui库，作为一个轻量级和通用图形库，它提供了创建嵌入式GUI所需的一切。具有易于使用的图形元素，美丽的视觉效果和低内存的占用。

Lvgl gui库的主要特点：

* 具有非常丰富的内置控件，像buttons，charts，lists，sliders，images等。
* 高级图形效果：动画、反锯齿、透明度、平滑移动。
* 支持多种输入设备：touchpad，mouse，keyboard，encoder等。
* 支持多语言的UTF-8编码
* 支持操作系统、外部存储和GPU（非必须）
* 支持单帧缓冲设备显示高级视觉特效

以上是lvgl支持的主要功能，但往上提供给应用开发还需再封装一层，例如控件层之类的。用以减少应用程序开发的工作量。

# 摄像头子系统

通过spi连接外部摄像头子模块，主要应用于二维码识别或者拍照静态照片的场景。系统需要支持opencv（用于识别）和ffmpeg（用于照片存储）等第三方库。

# OTA子系统

# 音频子系统

支持音乐播放、电话通话等功能，音频需支持输出和输入。但音频编解码由硬件完成，属于sensor hub模块。这块需要SC BU给出音频编解码方案，包括数据共享和交互。

# 电话子系统

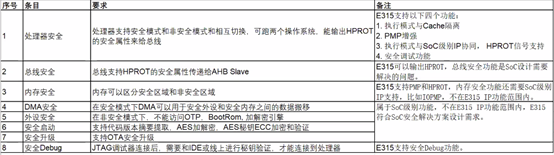
硬件模块属于SC BU，需要SC BU出解决方案

# sensor子系统

硬件模块属于SC BU，需要SC BU出解决方案

# 安全子系统

安全包括硬件安全实现方案和软件实现方案。目前E315的安全只承若下列表格的1、8项。



# 扩展子系统

## 蓝牙子系统

硬件模块属于SC BU，解决方案需要SC BU提供。

## AI子系统

## Cloud子系统

# 术语、定义和缩略语

## 术语、定义

本文使用的术语、定义如表所示：

# 参考文献

1、<https://help.aliyun.com/document_detail/308356.html> 阿里云OS

# 问题

1. 安全模块的方案还不确定。
2. 电话子系统方案不确定。
3. AI和云子系统方案不确定。
4. Sensor hub和cat方案不确定。
5. 没有手表相关产品经验的架构师。
6. 有无触摸屏的资料。
7. 如果用freetype支持汉字输入，光字库就要10M左右。8M flash空间不够。
8. GUI的组件开发，涉及工作量较大。看能否有开源资料进行借鉴开发？