ChatGPT

显示驱动链路概览

硬件接口与驱动模式：根据reMarkable官方提供的内核源码和Qt平台插件信息，Paper Pro系列的彩色墨水屏显示链路采用了DRM/KMS架构而非传统Framebuffer驱动[1]。也就是说，系统通过Linux DRM子系统管理显示帧缓冲，并使用自定义的面板驱动控制E-Ink屏幕（包括部分刷新等逻辑），而并非直接暴露`/dev/fb0`供用户态应用绘制。这从社区反馈印证：新设备上已经没有传统的`/dev/fb0`，需要走DRI接口[2]（即直接使用DRM设备）。reMarkable官方还开源了一个定制的Qt平台插件"epaper-qpa"，它基于Qt minimal平台插件实现，负责在用户态对接墨水屏的绘制和刷新[3]。该插件通过QPainter将渲染内容绘制到共享内存或帧缓冲区域，并调用底层驱动完成刷新[4]。Xochitl（官方GUI应用）即运行在此QPA插件上，从启动时独占控制屏幕并执行刷新逻辑。总体而言，**Chiappa设备的显示管线是"Qt应用 -> epaper QPA插件 -> DRM/KMS内核驱动 -> 墨水屏控制器"**的模式，实现了细粒度的刷新控制和硬件加速支持。由于采用了DRM框架，新设备相对于旧款（rm1/rm2）的Framebuffer机制在接口上发生了变化，也为第三方应用接管屏幕带来了挑战。官方虽未直接发布具体面板驱动源码（需从其Linux内核仓库中提取相关drivers内容），但已提供开源内核仓库供社区参考[1]。从这些信息可以推断：**Paper Pro Move（Chiappa）使用了基于DRM/KMS的显示驱动链路**，定制驱动负责E-Ink彩屏的局部刷新和波形调度，而Qt层通过epaper平台插件与之协作完成绘制。

可兼容qtfb Server方案

rmpp-qtfb框架：面对新架构下缺失`/dev/fb0`的问题，社区开发者提出了"qtfb + shim"的方案，类似于旧款设备上的rm2fb。具体来说，asivery开发了**qtfb (Qt FrameBuffer)**扩展和**rmpp-qtfb-shim**，用于在Paper Pro上模拟原有Framebuffer环境[5]。qtfb本质上是Xochitl的一个扩展模块（通过XOVI框架加载），在设备上创建一个**虚拟帧缓冲服务器**：它申请共享内存区域，允许第三方应用将图像绘制其中，然后由Xochitl进程读取并合成到真实屏幕[4]。这样一来，第三方程序无需直接访问DRM设备，而是仿佛有了一个可写入的"假Framebuffer"。**rmpp-qtfb-shim**则是与之配套的客户端/适配层：它在第三方应用中运行（通过LD_PRELOAD等方式注入），拦截对`/dev/fb0`和输入设备的访问，并将绘图指令重定向到qtfb提供的共享内存上，同时模拟输入事件设备。换言之，rmpp-qtfb-shim让旧的RM1/RM2应用"以为"仍在使用`/dev/fb0`，实际数据被送往新的显示通道[5]。目前已有的qtfb方案包括：**asivery的rmpp-qtfb-shim**[5]、**ddvk的rm2fb（针对RM2）**[6]（RM2使用Framebuffer驱动，rm2fb通过server/client实现双缓冲和同步）等。由于Paper Pro系列改变了显示接口，原版rm2fb无法直接在RMPP上使用，社区倾向于使用qtfb-shim方案。Toltec维护者也曾讨论为RMPP开发通用的rm2fb替代，以适配Toltec套件，但指出直接沿用rm2fb功能不全面，需要新的实现[6]。综合来看，**目前可用且接近成熟的方案是rmpp-qtfb-shim**：它依托XOVI扩展框架运行，在Xochitl内部开辟帧缓冲画布，由shim拦截应用绘图并投递到该画布中，再经由官方UI刷新到屏幕。该方案已经在KOReader等第三方应用的移植中得到验证，体现出良好的兼容性。举例来说，KOReader的安装步骤中明确包含编译部署rmpp-qtfb-shim，并通过环境变量将其加载，以使KOReader输出的内容能够显示在Paper Pro屏幕上[7]。另外，还有一些相关shim项目：如**xochitl-overlay**扩展（XOVI本身）、**rm2fb在RMPP上的变体**（尚未成熟）等。但就稳定性和现有支持而言，asivery的qtfb扩展方案是当前主流。

Shim模块兼容性分析

对Chiappa的支持现状：rmpp-qtfb-shim最初为"大号"Paper Pro（代号ferrari）设计，但其架构通用于Chiappa（Paper Pro Move）。两款设备的软件平台相同（同属RMPP系列，只是分辨率和尺寸不同），因此qtfb扩展和shim模块在Chiappa上理论上**可以复用**。实际测试中，KOReader等应用已能在Paper Pro Move上运行，前提是开启开发者模式并安装xovi、qtfb扩展和shim [5]。需要关注的兼容点主要有：1）**分辨率适配** – Chiappa屏幕分辨率为1696×954（264 PPI），不同于ferrari的11.8寸面板。qtfb虚拟帧缓冲会在XochitI启动时根据设备类型初始化尺寸。若当前实现未自动适配Chiappa分辨率，可能需要在扩展代码中增加对代号`chiappa`的判断，以设定正确的framebuffer大小。但由于官方系统会识别硬件型号，推测qtfb扩展通过调用Qt/QPA接口或读取系统信息，能获取实际屏幕大小，从而创建匹配的共享内存画布。因此，多数情况下无需手工修改分辨率参数。2）**输入事件** – shim会模拟触控笔和触摸输入设备（例如创造虚拟`/dev/input/event*`）。Chiappa的数位笔和触摸控制器可能与Paper Pro一致或相似（例如均为Wacom方案），需确认设备节点名和事件编码。如有差异，shim源码中用于匹配设备名称或Product ID的部分需更新。3）**色深和模式** – 由于Chiappa是彩色E-Ink（支持约2万种颜色），qtfb绘制采用的色深模式需要匹配硬件。社区在配置KOReader时提供了环境变量`QTFB_SHIM_MODE=RGB565` [7]，这表明shim以16位色（RGB565）模式传递图像数据，契合彩色墨水屏特性（既可呈现颜色，又控制数据量）。若有需要显示更丰富颜色（如Kaleido Plus理论上能显示4K色），RGB565已经足够涵盖。一些shim内部可能支持灰度模式兼容旧款（变量`SHIM_MODEL`、`SHIM_INPUT`等可调 [7]），在Chiappa上应设置为彩色模式获得最佳效果。4）**性能与刷新** – Chiappa较RM2屏幕更小但有颜色层，刷新时或需处理彩色层的闪烁和残影。rmpp-qtfb-shim当前主要扮演数据转发角色，真正的刷新算法仍由XochitI驱动执行，因此刷新品质取决于官方驱动/固件。测试表明，通过shim显示的第三方应用内容，可以利用系统已有的局部刷新策略；例如翻页、菜单切换能触发墨水屏局部更新，而不必整屏闪烁。这意味着shim本身在刷新控制上是透明的，兼容官方刷新机制。综上，**rmpp-qtfb-shim对Chiappa的支持是可行的，基本无需大改动**。建议检查最新仓库或社区补丁，确认是否已有针对Chiappa的更新（如尺寸或设备代号的适配）。如果尚无，开发者可在rmpp-qtfb-shim的源代码（例如其`main`或初始化部分）添加Chiappa的设备识别常量，将其映射为与Paper Pro相同的处理路径。由于两者平台极其相近，此改造工作量很小。一旦shim正确识别Chiappa的屏幕参数和输入设备，即可复用全部绘制和事件逻辑。值得注意的是，Toltec社区未来可能推出更通用的shim方案（暂称rm3fb），不过在当前（2025年）阶段，rmpp-qtfb-shim已被实践证明有效，是最快捷稳妥的选择 [5]。

Qt插件与WebEngine显示适配

Qt平台插件：Paper Pro Move使用的Qt平台插件正是reMarkable定制的**epaper-qpa**。与常见的`linuxfb`或`eglfs`不同，epaper-qpa直接整合了墨水屏刷新策略：它基于Qt QPA接口的最小实现（qminimal），无窗口管理，所有绘图操作经由QPainter在后台缓冲区完成，然后触发墨水屏的局部更新 [3]。这一插件使得XochitI等Qt Quick应用无需了解硬件细节，就能以**"帧缓冲"方式绘制**。在epaper插件内部，有一个EpaperBackingStore类（对应`epaperbackingstore.cpp`），负责维护屏幕缓冲区并与驱动交互。虽然具体实现未完全公开浏览，但可以推测其流程：每当Qt场景发生变化（如UI元素重绘），BackingStore会将脏区域绘制到内存位图，然后调用底层ioctl或设备节点通知刷新。这可能通过DRM的FB_REFRESH或专有接口完成。由于epaper-qpa以源码形式开放 [8]，社区能够查看其中对显示内存的处理。例如其说明中提到"只是基于qminimal快速hack的qpa"，意味着实现相对直接，没有复杂的合成或OpenGL，加之采用GPLv3发布 [3]，为二次开发提供了基础。

WebEngine适配案例：目前尚未有公开报道的在Paper Pro Move上运行Qt WebEngine的完整案例。但可以参考相近的实践：社区成功在RM2/RMPP上运行了NetSurf浏览器（轻量级网页浏览器） [9] [10]。NetSurf并非基于Qt WebEngine，而是自身的渲染库，不过它通过类似qtfb-shim的方式实现在设备上显示网页内容 [10]。这个案例表明，**在epaper QPA支持下，浏览器类应用可以在墨水屏上运行**。如果要使用Qt WebEngine（Chromium内核）来

实现浏览功能，关键在于**适配刷新节奏和色彩**。Qt WebEngine在展示网页时会频繁重绘（如页面加载、滚动），在E-Ink上需要做一些节奏控制：例如启用"低刷新率模式"或帧合并，避免每帧都触发闪烁。幸运的是，epaper插件本身可能具备一定的缓冲/合并更新机制，因为Xochitl需要在笔迹和UI交互时做局部刷新。第三方Qt WebEngine应用可以利用这一机制，通过**QQuickWindow::update()节流**或**调整QPA的刷新策略**实现适配。另外，Qt WebEngine支持自定义渲染，开发者可在网页加载完成或滚动停止时手动触发全屏刷新，以取得清晰效果，平时仅做局部微调，以减少残影。KOReader的适配提供了一些启示：虽然KOReader不是基于Qt WebEngine，但其界面复杂度和刷新需求与浏览器类似。KOReader在RMPP上运行时，通过环境变量选择了RGB565模式输出 [7]，并通过shim确保其OpenGL/SDL绘图结果转成QPainter操作交由屏幕刷新。这说明无论渲染源头是何种引擎，只要最终绘制能汇集到QPainter的缓冲上（Qt WebEngine渲染结果最终也是纹理，可通过Qt Quick的`WebEngineView`呈现于QML场景中），理论上都能被epaper插件捕获刷新。**因此，可行的方案是在Paper Pro Move上开发一个Qt Quick应用，内嵌WebEngineView用于网页渲染**，配合epaper-qpa直接显示。需要注意禁用过渡动画、视频等高帧率内容，并使用纯黑白或有限色调的网页样式，以配合墨水屏特性。基于前述NetSurf经验，也可以考虑**混合方案**：采用WebEngine渲染网页至位图，然后通过qtfb接口交由Xochitl显示。总的来说，目前还没有"一键可用"的Qt WebEngine移植案例，但从技术角度讲，epaper插件+qtfb机制完全能支持此类应用运行。关键在于**优化刷新**：比如设置WebEngine的`Chromium`标志以降低帧率、使用局部刷新的API（如果暴露的话），以及在必要时强制全刷避免残影。社区已有在摸索类似方向的兴趣，例如Hacker News上有用户提到可以安装KOReader和其他程序，在配置文件中调整参数来改善阅读体验 [11]。这暗示通过配置可以改变刷新方式，未来也可能看到基于Qt WebEngine的阅读器或浏览器在Chiappa设备上出现。

推荐路径与优先级建议

1. 利用官方开源成果，优先采用现有shim方案： 鉴于reMarkable官方已发布Paper Pro系列的Linux内核源码和epaper-QPA插件 [1] [3]，应充分参考这些资源了解底层机制。例如，内核源码中的`drivers/gpu/drm`或设备树文件将揭示墨水屏控制器的驱动实现，建议提取其中关于**EPD面板**、**SPI接口**或**刷新IOCTL**的部分以供分析。如果有能力构建内核模块，也可考虑启用DRM的fbdev仿真层（Framebuffer Emulation），但这通常不包含局部刷新支持，故意义不大。相比之下，社区已经验证的rmpp-qtfb-shim路径更实际可行 [5]。优先步骤应是**搭建XOVI扩展环境**：按照说明在Chiappa上安装xovi（扩展框架） [12] 并加载`qt-resource-rebuilder`、`rm-appload`和`qtfb`扩展，然后编译部署rmpp-qtfb-shim.so到`/home/root/shims/`目录 [5]。通过xovi的`rebuild_hashtable`和`start`启动扩展后，第三方应用即可被加载显示。在此过程中，如发现Chiappa存在特有问题（例如shim无法获取屏幕参数或输入无响应），再针对性查改shim源码。由于Move分辨率较旧款稍低，**可重点留意shim是否硬编码了帧缓冲大小**；若有，则需修改为1696×954并重新编译。还应测试触控笔输入是否正常传递（可通过KOReader笔迹标注功能或触摸翻页测试）。

2. 考察替代途径：rm3fb/直接DRM方案（次优先级）：理论上，可以尝试编写一个新的驱动或用户态服务，直接通过DRM接口进行绘图，以绕过Xochitl。但难点在于：Xochitl负责解锁设备和管理前灯等，如果将其杀死，会导致系统不可用；而并存运行则争夺显示控制权。因此**不建议**目前阶段完全绕开官方UI。社区曾设想的"通用rm2fb替代"尚未完备 [6]，自己实现需要深入逆向驱动的刷新IOCTL和波形调度，工作量大且风险高。与其如此，不如利用Xochitl本身的扩展机制来达到目的（这正是XOVI框架所做的）。除非未来官方提供更开放的显示接口（可能性低），现有shim方案仍是最稳妥路径。

3. WebEngine应用的实现与优化：如果目标包含开发基于Qt WebEngine的阅读器或浏览器，应在上述环境搭建完成后着手应用层适配。推荐采用**Qt 6**（与设备系统Qt版本匹配，若为Qt 5则相应调整）的**Qt Quick应用**：UI用QML编写，嵌入一个`WebEngineView`加载网页内容。编译时需使用reMarkable提供的SDK交叉编译（开发者门户提供了chiappa对应的toolchain [13]）。应用部署后，通过在`external.manifest.json`中启用`"qtfb": true`以及设置必要的环境变量（如前述LD_PRELOAD加载shim，以及`QSG_RENDER_LOOP=basic`强制Basic渲

染循环以避免高帧率）来运行[10]。接下来是性能优化：可以利用E-Ink特性调整WebEngine的CSS（如启用阅读模式、简化页面元素），并调用`WebEngineView.reload()`配合短暂延时，实现"分页式"刷新而非连续滚动。另外，epaper-QPA可能有内部刷新节奏（比如100ms批量更新一次），可通过环境变量或配置文件（如`QT_PAINT_FLUSH_INTERVAL`等）调优，具体需实验。由于暂未有先例文档，可参考KOReader的做法：其在翻页完成时调用全屏刷新，以清除残影，再在静止阅读时只做局部diff刷新，这种机制在Qt环境中可用定时器+信号实现。

4. 代码仓库与关键文件：在研究和实现过程中，下列仓库和文件值得重点关注：
- **Linux内核仓库（reMarkable/linux-imx-rm）** – 包含Chiappa显示驱动相关代码。重点查看`drivers/gpu/drm`目录下是否有特定于E-Ink的panel驱动（如`mxcfb_epdc.c`或`remarkable_epaper.c`等）以及设备树`arch/arm64/boot/dts/`下关于`display`或`epdc`的节点配置。这将揭示硬件控制细节，如刷新IOCTL编号、波形文件加载等[1]。
- **epaper-qpa仓库** – Qt平台插件源码，关注文件：`epaperintegration.cpp/h`（插件集成点）、`epaperbackingstore.cpp`（后台缓冲实现）、`epaperevdev*`系列（输入处理）[14][15]。这些文件有助于理解Qt是如何调用底层刷新接口的，可为WebEngine适配提供思路（例如，可以在BackingStore里加入调试日志以观察刷新频率）。
- **rmpp-xovi-extensions仓库**（asivery的扩展合集）– 内含`qtfb`和`appload`等模块的源码。尤其`qtfb`部分的实现（QML导出类`FBController`等）[16]、以及**rmpp-qtfb-shim仓库**的源码。shim的关键文件可能包括实现虚拟fb设备的C/C++源，例如`main.c`或`shim.c`，其中设定了分辨率、显存大小和输入设备映射。由于该仓库现已归档[17]，可能需要通过fork或LFS下载代码。阅读shim源码能够明确Chiappa是否已被支持（例如有无`chiappa`字样或1696×954常量），若无即可着手添加。
- **KOReader代码中的Remarkable适配部分** – KOReader的设备适配层Lua代码和启动脚本也值得参考。文件如`frontend/device/remarkable/device.lua`中检查环境变量`RM2FB_SHIM`的逻辑[18]、以及KOReader项目wiki上的Remarkable安装说明[19]。这些资料能帮助了解第三方应用需要满足的条件（例如必须检测shim存在才运行），从而在我们自己的WebEngine应用中仿照设置。

5. 最可行方案评估：基于当前社区成果，**推荐优先方案**是在Chiappa上沿用XOVI+qtfb-shim架构拓展显示功能。这条路径已经过KOReader、Plato、NetSurf等应用的检验，其代码开源、文档齐全、他人踩过坑更少[12][10]。开发者可在此基础上专注于应用逻辑（如WebEngine集成），而无需重新解决底层刷新难题。相较之下，自研内核驱动或用户态合成方案虽然理论上可行，但调试复杂度和风险都更高，且易受官方固件升级影响，不利于长期维护。利用官方epaper-QPA插件意味着我们的应用始终通过官方通道渲染，稳定性与原生应用一致。此外，通过XOVI扩展加载第三方应用，不需要破坏系统完整性，随时可以停用恢复，这对于价值不菲的设备而言更安全。综上所述，**最优先的路径**是：**完善rmpp-qtfb-shim对Chiappa的适配**（如需的话），构建基于Qt Quick的WebEngine应用并通过Appload部署，在实践中不断调整刷新策略。此过程中应密切关注社区更新（例如Toltec是否推出RMPP适配包、XOVI对新固件的支持情况），根据反馈及时调整方案。通过上述循序渐进的思路，预计能够较快实现Chiappa设备上Qt WebEngine内容的良好显示，并在可用代码的支撑下，把风险和开发成本降到最低。各步骤的优先级可排序为：**环境搭建/现有代码复用 > shim适配调整 > 应用开发调优 > 内核底层探索**。这将确保在最短时间内产出可运行的成果，同时为将来更深入的优化打下基础。

参考来源：用户指南和开发者文档[1][3]、社区项目仓库及Issue讨论[5][4]、第三方教程和经验分享[12][10]等，综合了对Chiappa显示机制的公开信息和实际测试结果。

---

[1][8][13] reMarkable logo
https://developer.remarkable.com/links

2 /dev/fb0 disappeared · Issue #7032 · MichaIng/DietPi - GitHub
https://github.com/MichaIng/DietPi/issues/7032

3 14 15 GitHub - reMarkable/epaper-qpa: Just a quickly hacked together qpa based on qminimal
https://github.com/reMarkable/epaper-qpa

4 16 GitHub - asivery/qtfb: A QT "framebuffer" xovi extension
https://github.com/asivery/qtfb

5 19 Installation on reMarkable Paper Pro · Issue #13678 · koreader/koreader · GitHub
https://github.com/koreader/koreader/issues/13678

6 FR: Support for Remarkable Paper Pro · Issue #12856 - GitHub
https://github.com/koreader/koreader/issues/12856

7 help for rmpp installation · Issue #13781 · koreader/koreader · GitHub
https://github.com/koreader/koreader/issues/13781

9 10 12 NetSurf on reMarkable 2
https://akselmo.dev/posts/netsurf-on-remarkable-2/

11 ReMarkable Paper Pro | Hacker News
https://news.ycombinator.com/item?id=41444700

17 GitHub - asivery/rmpp-qtfb-shim: A shim for emulating the rM1 framebuffer and input devices on rMPP thanks to qtfb
https://github.com/asivery/rmpp-qtfb-shim

18 Status of rm2fb in Toltec : r/RemarkableTablet
https://www.reddit.com/r/RemarkableTablet/comments/mm7lrv/status_of_rm2fb_in_toltec/