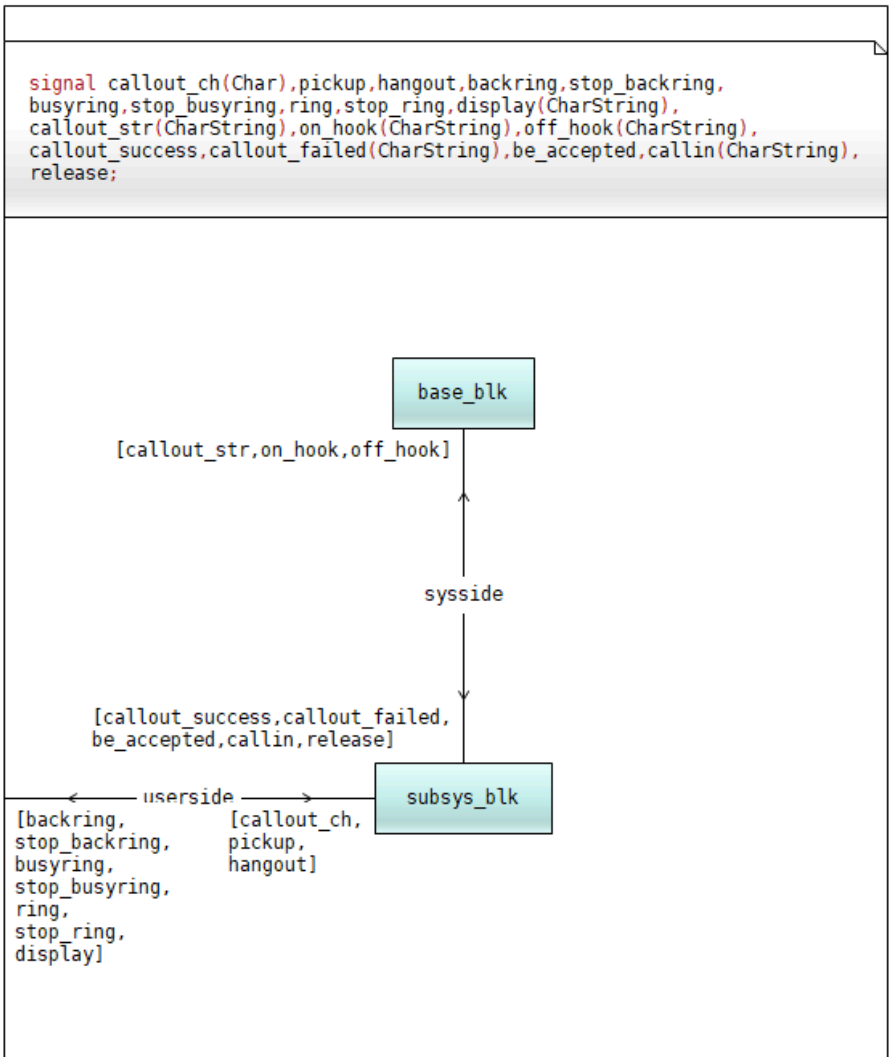
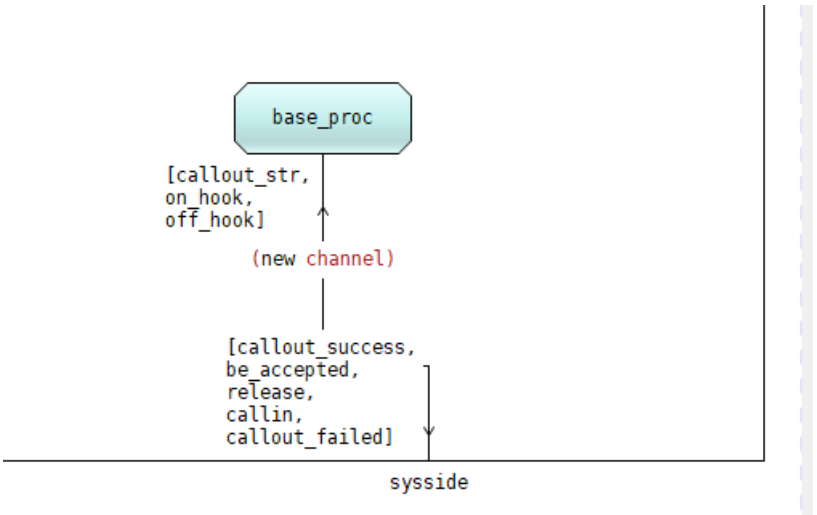
Phone设计思路



在系统级别的设计图中，通过将整体系统分为“系统侧（sysside）”与“用户侧（userside）”来进行抽象。其中，`base_blk`、`subsys_blk`等功能块负责在系统层面进行信号的初步分发与处理，为后续的进程（process）提供清晰的消息入口和出口。通过这些块的定义和接口信号的配置，确保系统在宏观结构上被合理划分，使得信号在系统层面流动清晰、无歧义。

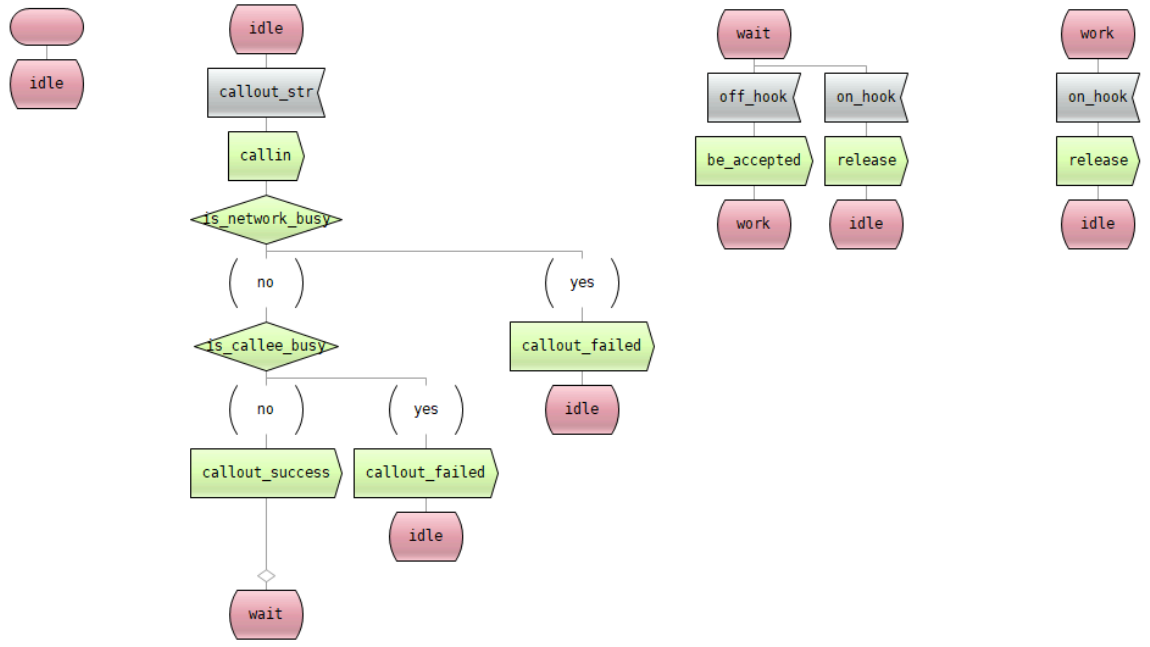
base_blk



`base_blk` 是系统最高层级的基础功能块之一，主要用来与系统外部（如用户界面、底层硬件平台或外部资源）进行初步交互。它在系统与环境之间充当一个门面接口，将来自系统外部的信号（如拨号字符串 `callout_str`、用户摘机 `off_hook`、挂机 `on_hook` 等）吸收，并将其转换、过滤或直接传递给系统内部更具体的进程。

通过 `base_blk` 的存在，系统获得了一层抽象，使后续子系统或进程只需专注于内部流程逻辑而不必关注外部环境的复杂性，从而提升系统的可维护性与模块化程度。

base_proc



`base_proc` 作为系统内处理基本呼叫逻辑的核心进程之一，主要负责在呼叫建立的初始阶段与后续基本事件中根据外部输入（如拨号请求）、系统资源状态（如网络忙/闲、被叫忙/闲），以及用户操作（如摘机off_hook、挂机on_hook）来决策和调度关键信号。下面分阶段对其进行介绍：

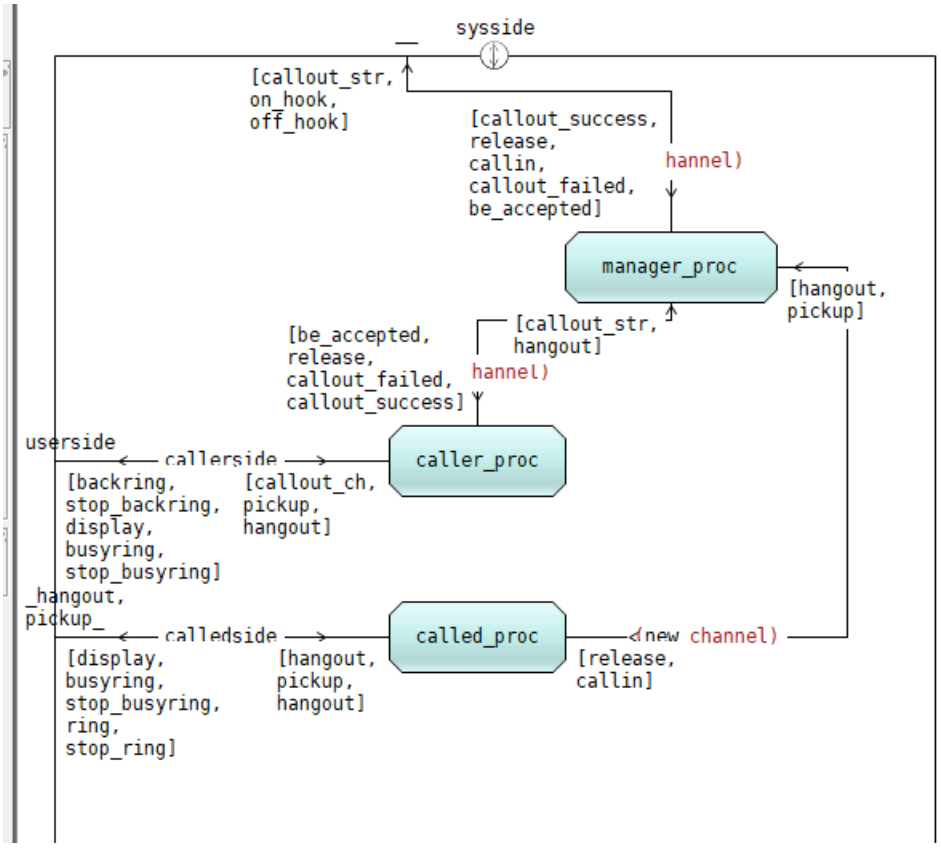
1. **初始状态与拨号请求处理**
   在 `base_proc` 的状态图中，初始状态为 `idle`，表示系统处于空闲等待状态。当从 `base_blk` 接收到用户发起呼出请求的信号（`callout_str`）时，`base_proc` 开始对呼叫进行尝试性建立。在这一阶段，它会发送 `callin` 信号，将呼叫请求传递给内部的流程管理或资源调度模块（如 `manager_proc`）。
2. **资源检查与决策**
   接收到 `callout_str` 后，`base_proc` 不会立即进入已建立呼叫的状态，而是先检查系统资源的可用性，即通过判断 `is_network_busy` 与 `is_callee_busy` 的条件分支来决定接下来的动作：
   - **网络是否忙（`is_network_busy`）：**
     若网络忙，则无法建立呼叫。此时 `base_proc` 会发送 `callout_failed` 信号并返回 `idle` 状态。
   - **被叫是否忙（`is_callee_busy`）：**
     若被叫已忙线，同样无法建立呼叫。`base_proc` 这时同样发送 `callout_failed`，并回到 `idle`。
   - 若网络和被叫均不忙，则表示呼叫建立条件满足，`base_proc` 会发送 `callout_success` 信号，表示呼叫资源已成功分配。此时状态机进入 `wait` 状态，表示系统已等待被叫端响应或用户的进一步操作。

3. **等待状态与用户接听/挂机处理**
   在进入 `wait` 状态后，`base_proc` 等待用户的摘机（off_hook）或挂机（on_hook）等进一步操作：
   - 若 `off_hook` 信号到来，意味着被叫或用户准备接听，这时 `base_proc` 会发送 `be_accepted`，表示通话接入已获得确认，并将状态转移至 `work` 状态（工作态）。此状态下表示通话正在进行或已准备就绪。
   - 若在 `wait` 状态下收到 `on_hook` 信号，表示主叫或被叫在建立通话前就选择挂机，则 `base_proc` 会发送 `release`，释放资源，并将状态返回 `idle`，通话未进入工作阶段。
4. **工作状态与结束通话**
   当 `base_proc` 进入 `work` 状态，表示呼叫已成功建立，用户处于通话中或已准备充分的工作阶段。在这个阶段，如若用户发送 `on_hook`（挂机）信号，则需要释放资源（`release`），并将状态返回 `idle`。
   `work` 状态通常是建立的呼叫完成后的稳定运行态，也是通话结束前的最后状态，一旦通话结束（挂机），`base_proc` 将回归 `idle`，等待下一个呼叫请求的到来。

整体而言，`base_proc` 的作用就是将呼叫建立过程流程化：

- 从空闲（`idle`）开始，当用户请求拨号（`callout_str`）时，进行网络及对端资源检查。
- 若条件满足则建立呼叫（`callout_success`）并转入等待状态（`wait`）。
- 在等待状态中根据用户接听或挂机决定进入工作态（`be_accepted` -> `work`）或直接释放回到空闲态（`on_hook` -> `release` -> `idle`）。
- 在工作态下通话进行，一旦用户挂机再次释放资源并回到 `idle`。

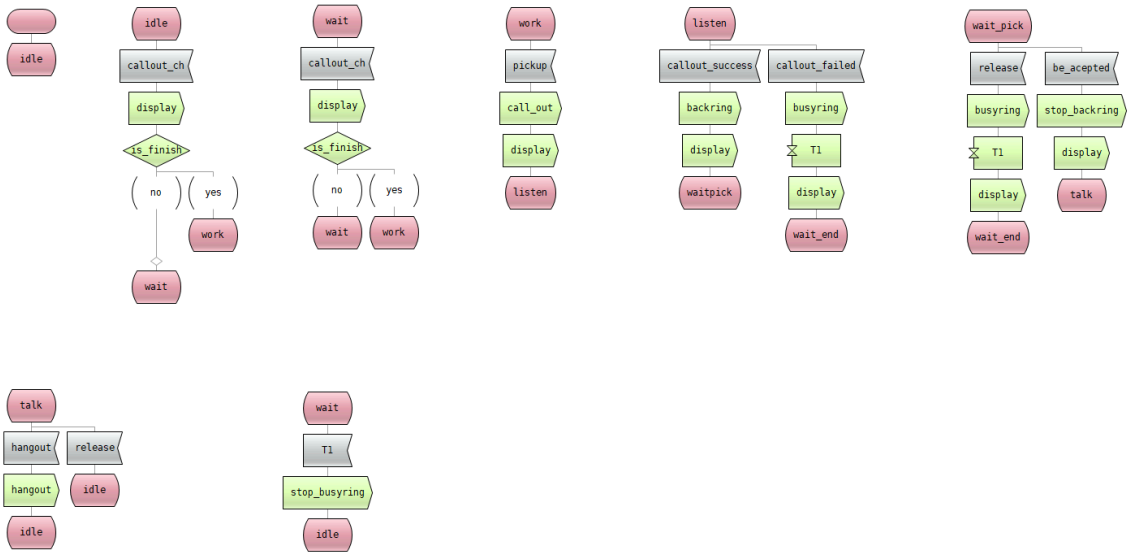
subsys_blk



`subsys_blk` 是系统内相对底层的功能块，用于进一步细分功能逻辑和信号路由。与 `base_blk` 相比，`subsys_blk` 更接近用户逻辑过程（如呼叫流程控制、信号发送与接收、显示与振铃控制等）。它承担了对用户侧（userside）事件、信号的解释与传递工作，通过与上层块（`base_blk`）和内部进程交互，使系统内部的 `caller_proc` 和 `called_proc` 能够接收正确的信号并进行相应状态变化和动作执行。

`subsys_blk` 在分层结构中扮演中间调度角色，保证上层模块的抽象指令能正确落地到具体的用户行为控制流程中。

caller_proc



`caller_proc` 进程用于描述主叫用户在整个呼叫发起和通话流程中所经历的状态变化和响应的动作。通过状态图，我们清晰地定义了当用户从初始无呼叫状态到输入号码、发起呼叫、等待应答、接通通话、挂断结束的完整过程。下面将按状态流转与信号处理逐步介绍：

1. **初始与拨号阶段（idle -> callout_ch -> display）**
   在初始状态 `idle` 中，主叫用户处于空闲等待。如果用户开始输入拨号字符（`callout_ch`），`caller_proc` 会从 `idle` 转移到拨号处理状态，一旦接收到拨号输入信号，系统会触发 `display` 来显示已输入的号码或提示用户输入过程的状态。

   此时有一个分支条件 `is_finish`，表示用户是否完成号码输入：

   - 若尚未完成输入（no），则进入 `wait` 状态，继续等待用户输入更多字符或命令。
   - 若已经完成输入（yes），则表示用户输入号码完毕，`caller_proc` 进入 `work` 状态，为后续的正式呼叫发起作准备。

2. **发起呼叫与等待响应（work -> call_out -> display -> listen）**
   在 `work` 状态下，用户摘机（pickup）后发出正式的呼叫请求（`call_out` 信号）。此时系统会执行 `display` 来显示当前呼叫状态（例如正在呼出）。然后进入 `listen` 状态，表示已将请求发送给网络与对方用户，等待系统侧的反馈（`callout_success` 或 `callout_failed`）。

3. **处理呼叫结果（listen -> waitpick / wait_end）**
   在 `listen` 状态中，`caller_proc` 将根据系统反馈进行状态变迁：

   - 若收到 `callout_success`，表示呼叫成功接通，对方电话线路可用。此时会触发 `backring`（回铃音）和 `display` 显示此状态，同时转入 `waitpick` 状态，表示等待对方接听（或等待用户的进一步行动）。

- 若收到 `callout_failed`，表示呼叫失败（例如对方忙线或网络忙），则 `caller_proc` 触发 `busyring` 和 `display` 来显示忙音和相关信息，并根据内部计时器（`T1`）处理，在计时器到期时转入 `wait_end` 状态，以便用户在失败后返回到空闲状态或尝试其他操作。

4. **等待接听与通话建立（waitpick -> be_accepted -> display -> talk）**
   在 `waitpick` 状态中，如果对方接听（`be_accepted`），`caller_proc` 会通过 `display` 来显示呼叫成功进入通话状态，并转入 `talk` 状态。此时双方处于通话中，用户可以进行正常通话操作。

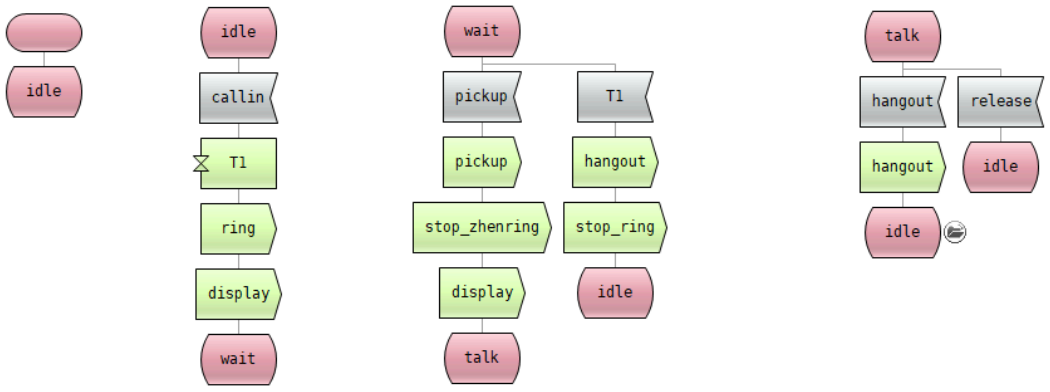
   同时，在 `waitpick` 状态下，若对方没有及时接听，系统可能超时或用户放弃，可以通过相应事件（如 `release`）重回空闲或结束状态。

5. **通话进行与结束（talk -> hangout -> release -> idle）**
   在 `talk` 状态中，用户和被叫正在通话。当用户主动挂机（`hangout`），`caller_proc` 将发送 `release` 信号以释放呼叫资源，同时将状态回归 `idle`，表示通话终止并返回到无呼叫状态。

   此流程确保一旦通话结束，资源回收与状态重置有序进行，使系统能够随时准备下一次呼叫。

6. **其他状态与事件处理**
   除上述主流程外，`caller_proc` 还处理其他中间状态或异常情况。例如在等待状态下用户可能中途放弃输入或挂机、在回铃过程中用户可能取消呼叫、在拨号后未能成功转入通话的情况下需要回退到空闲状态等等。这些通过 `display`、`busyring`、`stop_busyring`、`T1`（定时器）等信号和动作进行配合与控制，保证各种场景下用户体验的一致性与流程的完整性。

called_proc



`called_proc` 用于描述被呼叫用户侧（被叫端）在接到来电、选择是否接听以及挂断的整个状态转移流程。通过 `called_proc` 的状态图，我们可以清楚地看到从初始无呼叫状态，到被动接收到呼叫请求，再到被叫应答或拒绝，直至最终恢复到空闲的完整过程。下面将对各个关键状态和信号处理进行说明。

1. **初始状态与来电（idle -> callin）**
   当被叫用户处于 `idle` 状态时，说明当前电话处于待机没有来电或正在通话的状态。当系统收到来电请求信号（`callin`）后，`called_proc` 从 `idle` 状态转入处理来电状态。在此时，被叫收到一个呼叫请求，但尚未提示用户。这一阶段可以启动内部定时器（`T1`）来等待被叫用户的接听动作。

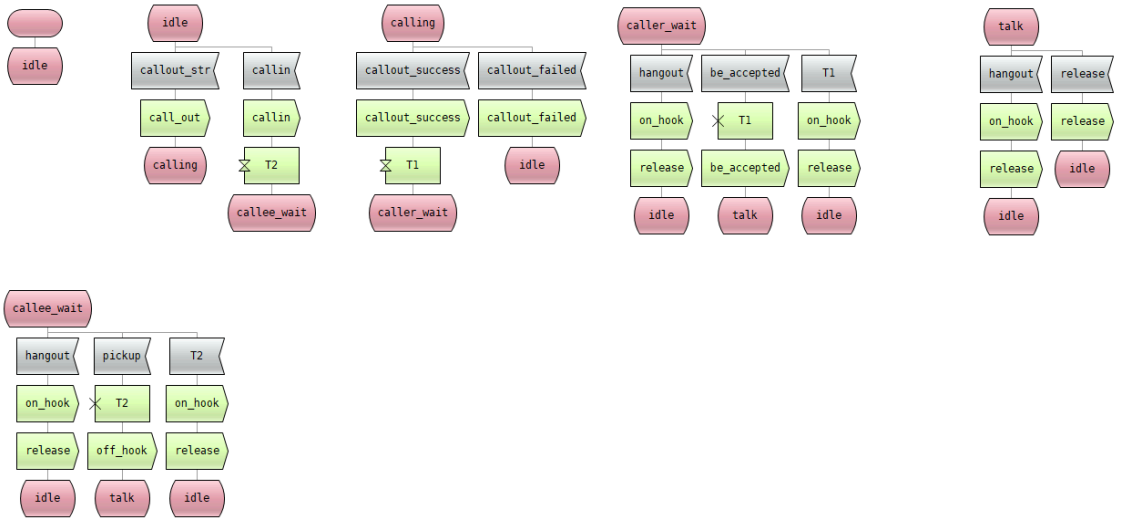
2. **来电提示与等待用户应答（callin -> ring -> display -> wait）**
   收到 `callin` 后，`called_proc` 通过 `ring` 信号触发手机响铃提示（例如振铃声），并使用 `display` 信号来在屏幕显示来电信息（如来电号码或来电标识）。完成这些动作后，状态机进入 `wait` 状态，表示此时正在等待用户采取实际动作（如接听 `pickup` 或挂断 `hangout`），用户端听到铃声并看见来电显示。

3. **应答与进入通话（wait -> pickup -> stop_zhengring/stop_ring -> talk）**
   在 `wait` 状态中，如果被叫用户决定接听电话（`pickup`），`called_proc` 会发出 `stop_zhengring`（或 `stop_ring`）信号停止响铃，然后进入 `talk` 状态，表示成功应答来电并建立通话。
   此时，被叫与主叫已处于通话状态，双方可以正常对话。`talk` 状态是一个稳定的通话工作状态，直到用户结束通话为止。
4. **拒接或超时处理（wait -> hangout -> stop_ring -> idle）**
   若被叫在 `wait` 状态下选择不接听（`hangout`）或者超时计时器 `T1` 到期（表明用户在规定时间内未应答），`called_proc` 将发出 `stop_ring` 停止振铃，并回到 `idle` 状态。这代表被叫未接听此来电，呼叫请求结束。

   此时，与主叫侧的连接未建立或已中断，呼叫资源被释放。
5. **通话结束（talk -> hangout -> release -> idle）**
   当被叫已经接听并处于 `talk` 状态中，若用户在通话过程中选择挂机（`hangout`），`called_proc` 会发出 `release` 信号释放呼叫资源，然后状态流转回到 `idle`，再次进入待机状态，准备接受下一次呼叫。

   通过对通话结束状态的明确定义，确保了每次通话的生命周期有序完成，从拨入到挂断都能有清晰的资源释放与状态恢复。

manager_proc



`manager_proc` 是整个系统呼叫控制流程的中枢调度进程，它通过协调主叫、被叫和系统侧的各种事件与信号，确保呼叫流程有序进行并在必要时执行决策和资源分配/释放。以下从状态流转和信号处理的角度对其进行详细说明：

1. **初始阶段与呼叫发起（idle -> callout_str -> call_out）**
   在初始状态 `idle` 下，当 `manager_proc` 接收到来自主叫侧的拨号请求信号（`callout_str`）时，表示用户已完成号码输入并希望发起呼叫。此时 `manager_proc` 通过 `call_out` 动作开始尝试呼叫建立过程。
   `manager_proc` 负责将这一请求转发给合适的子流程（例如通知系统尝试向被叫端发起来电）。通过这种方式，`manager_proc` 在最初就确立了对整个呼叫流程的统筹权。
2. **呼叫尝试与资源检查（call_out -> calling -> callin -> ...）**
   当 `manager_proc` 进入 `calling` 状态后，它会尝试通过 `callin` 信号向被叫侧发起呼叫请求。在这个过程中，`manager_proc` 可能需要检查网络是否忙，或者被叫是否忙，通过内部条件判断（如 `is_network_busy`、`is_callee_busy`）和计时器（`T2`）来决定下一步动作。

如果呼叫建立条件满足并通过底层资源检查，`manager_proc` 会向主叫侧返回 `callout_success`，同时让呼叫流程进入 `caller_wait` 状态表示成功呼叫已在进行中。如果呼叫失败（`callout_failed`），则会通知主叫进程以恢复到 `idle`。

3. **等待对方应答与计时器管理（caller_wait 与 callee_wait）**
   在呼叫请求成功发出后，`manager_proc` 等待被叫用户的反应（通过 `callee_wait` 状态）。
   - 若在规定时间内（由定时器 `T2` 控制）被叫用户没有接听（`pickup`），`manager_proc` 会触发相关失败处理（如 `release`），将系统资源释放并返回空闲状态。
   - 如果被叫接听（`pickup`）或系统侧传来 `be_accepted` 信号，`manager_proc` 则进入通话状态（`talk`）。此时双方通话成功建立，`manager_proc` 在通话期间保持对状态的监督和调度权。
4. **通话进行与结束（talk -> hangout -> release -> idle）**
   当进入 `talk` 状态表示呼叫成功完成并进入稳定的工作状态。此时通话双方可以自由交流。如果通话结束（无论是主叫侧还是被叫侧挂断 `hangout`），`manager_proc` 会负责协调呼叫的释放过程（`release`）。
   `manager_proc` 在获取到挂机信号后，会执行资源回收（`release`）并将状态机回到 `idle` 状态，为下一次呼叫做好准备。
5. **定时器的使用与异常处理**
   在呼叫过程中，`manager_proc` 通过设置定时器（如 `T1`、`T2`）来约束等待时间。例如：
   - `T1` 用于等待对方接听或一些关键应答信号。若定时器超时，则认定呼叫失败并进行相应清理动作。
   - `T2` 可能在等待被叫用户回应时使用，若被叫长时间未应答，`manager_proc` 会强制终止呼叫流程并返回空闲。
6. **状态之间的信号交互**
   在整个呼叫生命周期中，`manager_proc` 的状态流转紧密依赖于从主叫侧（`caller_proc`）、被叫侧（`called_proc`）以及系统侧（`base_proc`）传来的信号。
   - 当主叫输入号码并请求呼叫时，通过 `callout_str` 和 `call_out` 来开始呼叫尝试。
   - 当被叫应答时，通过 `pickup`、`be_accepted` 信号来转入 `talk` 状态。
   - 当任何一方挂机（`hangout`）或资源需要释放（`release`）时，`manager_proc` 及时退出通话状态并清理。
   - 当呼叫失败（`callout_failed`），`manager_proc` 及时通知主叫侧恢复到空闲状态。