

**主要功能**

EcuM模块作为AUTOSAR中的标准模块，全称为（ECU State Management）。故名思义，指的就是ECU 的状态管理，不过需特别强调的是ECU上下电流程的状态管理，具体可以简单概括为以下五个方面的内容：

Startup 初始化流程状态管理；

ECU运行状态管理；

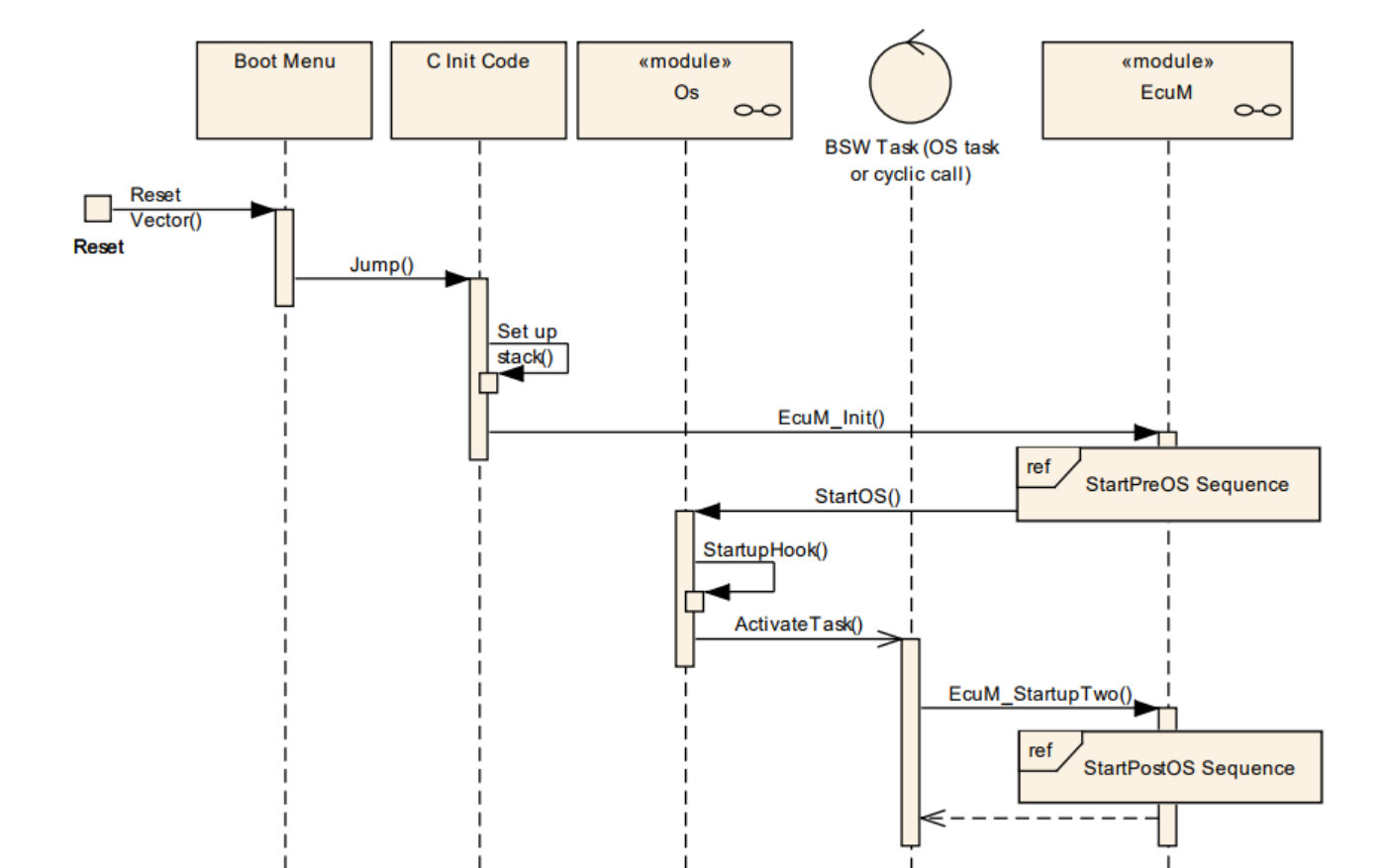
ShutDown流程状态管理；

Sleep流程状态管理

Wakeup Source管理；

**总状态机（Flexible 与 Fixed）**

在具体介绍上述5个状态管理过程之前，我们有必要对ECU启动过程有个总体的感性认识，以便于对后续各个阶段的之间的关系有个较为清晰的了解。如下图所示，描述了一般情况下ECU的启动流程。



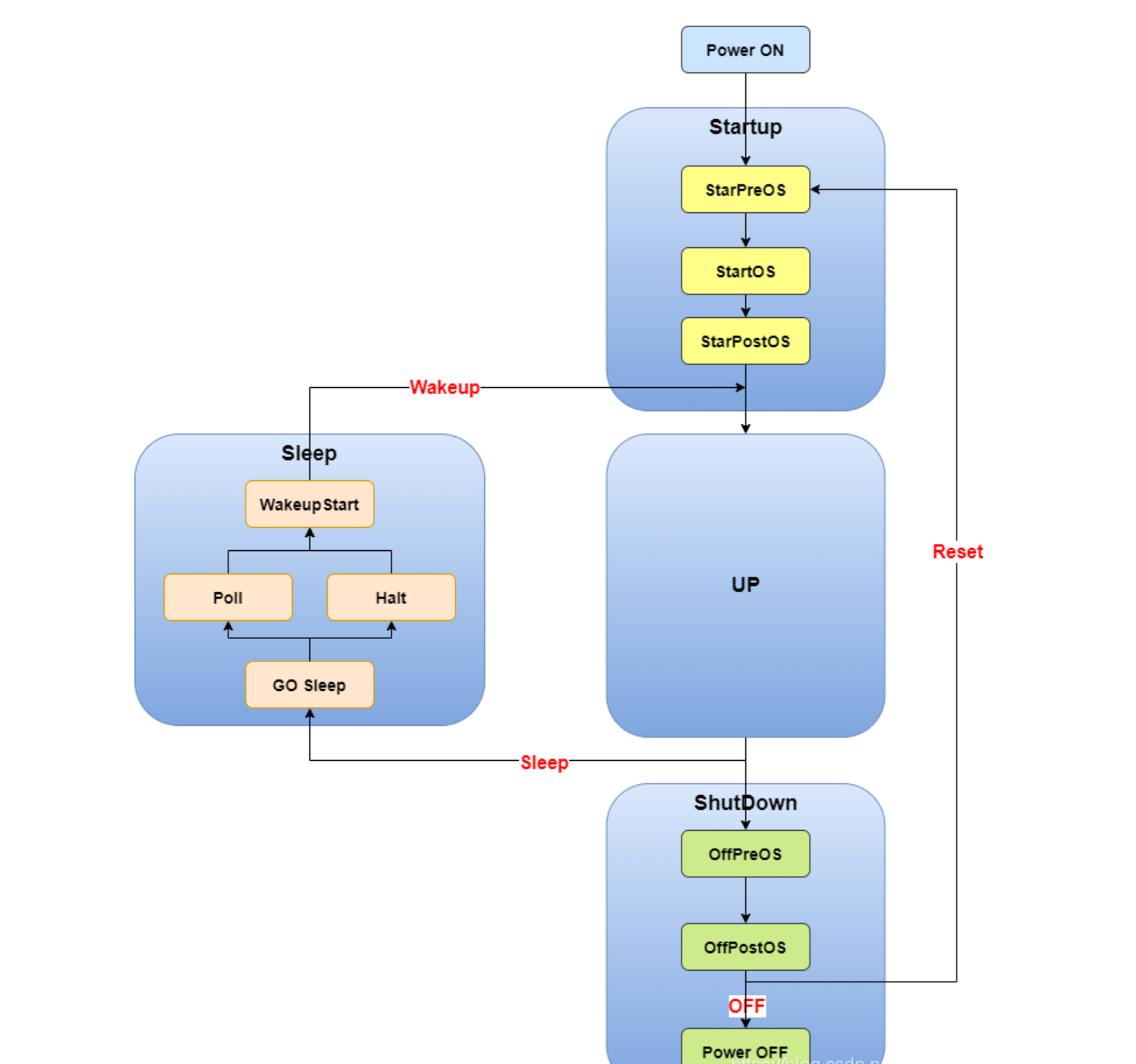
在上述的ECU启动过程中，可以看出ECU的一般启动过程涉及到Boot，C\_Init, EcuM，OS等模块，在这些模块的共同接力下保证BSW及RTE成功初始化，进而使得整个SW-C处于正常running的过程。

ECU启动时，首先通过中断向量表运行引导程序（俗称BootLoader），Bootloader在满足一定条件下跳转至APP程序中的C\_Init处并指向Main函数。

在Main函数中首先完成堆栈空间的初始化，然后调用EcuM\_Init函数进入到后续的StartPreOS，StartOS阶段，在开启OS的初始化函数中调用EcuM\_StartupTwo进行第二启动阶段的初始化，最后就是进入StartPostOS阶段，如完成BswM模块的初始化，进而将控制权转交给BswM模块。

由于接力赛中首棒很关键，因此将重点关注EcuM模块的启动与关闭过程，按照AUTOSAR定义，EcuM可分为两种模式：**Flexible与Fixed模式。**

**Flexible 总状态机，如下图所示：**



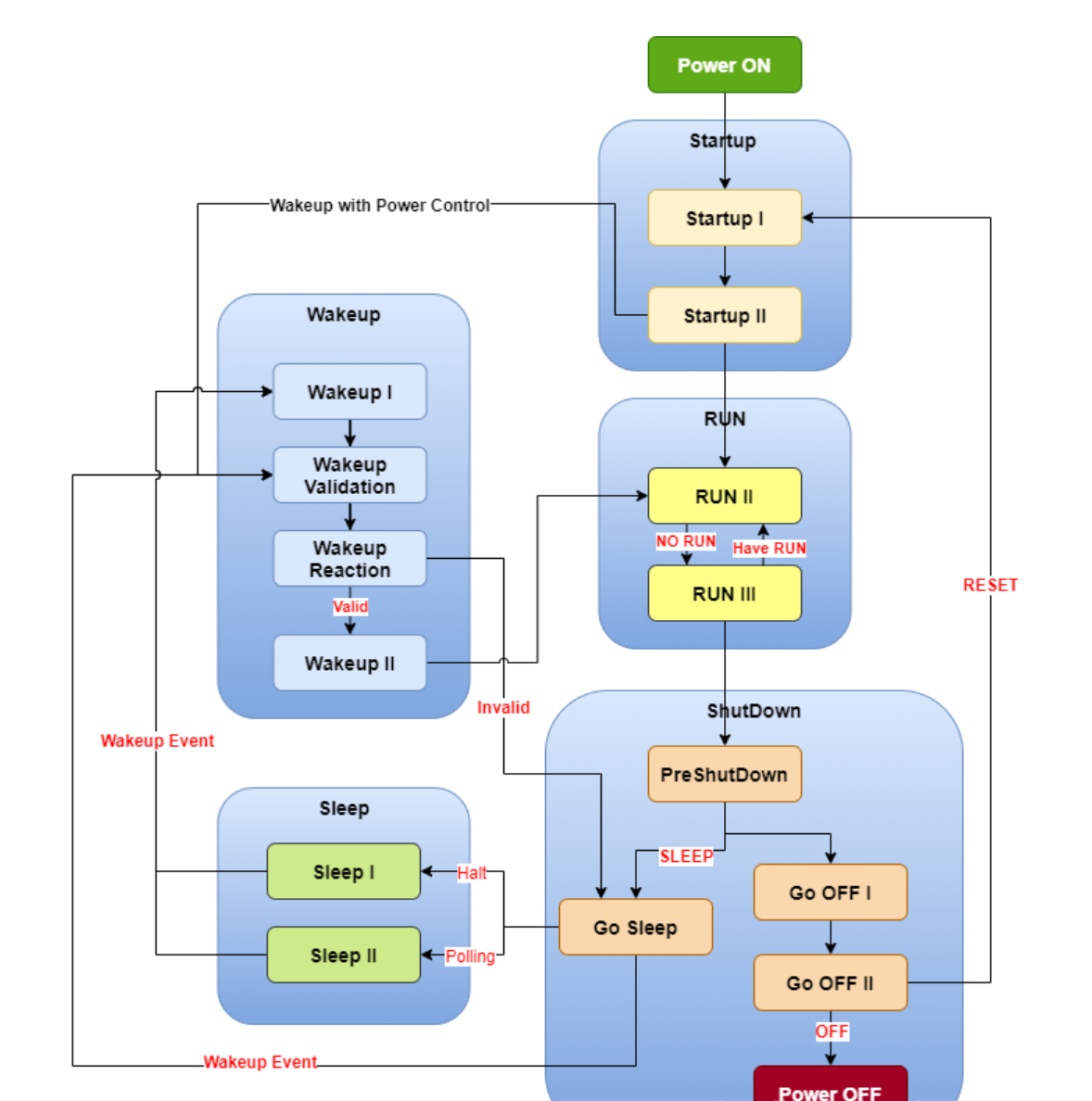
在上图中，Startup阶段按照开始OS节点作为分水岭，可分为StartPreOS与StartPostOS两个阶段。经历过Startup阶段之后，则会进入到UP阶段。

在UP阶段则是正常运行状态，当条件满足时，可以根据CPU是否进入到低功耗状态还是OFF状态，相应进入到Sleep阶段与ShutDown阶段，当然如果是Reset，那么也是先进入到Shutdown阶段，最后跳转至Startup阶段。

若进入到Sleep阶段之后，也存在着两种CPU低功耗模式：Poll与Halt模式，后者比前者更节约电能且无需运行代码，具体采用哪个则可根据当初的系统设计而定，在该阶段不会关闭OS，OS始终低功耗的running状态，同时也会不断的对唤醒源进行监控，若唤醒源满足，则会直接跳转至RUN阶段。

若进入到Shutdown阶段，会经历两个阶段：OffPreOS与OffPostOS阶段，前者则是为Shutdown OS之前所做的准备，后者则是关闭OS之后，选择对应的函数执行关闭ECU还是重启ECU的操作。

**Fixed 总状态机,如下图所示：**

****

在上图中，较为清晰的描述了EcuM Fixed模式下五种状态**Startup，Shutdown，RUN，Sleep，Wakeup**的状态组成以及状态切换的过程，其中**OFF，Sleep，RUN是稳态，而Startup跟Wakeup则是暂态**。

在Startup阶段，同样按照Flexible 模式中开启OS为界限，分为Startup I与Startup II两个阶段；当唤醒事件能够控制CPU供电时，则需要进入Wakeup阶段验证Wakeup Event是否有效，相反如果不带电源控制，则直接进入RUN阶段。

若进入到RUN阶段，可分为两个阶段：RUN II与RUN III两个阶段。其中RUN II指的是正常运行阶段，RUN III则是SW-C为即将进入到ShutDown所需要做的前提准备。

若进入到ShutDown阶段，首先会进入到PreShutDown阶段，然后按照Shutdown的目标不同，可以分为reset，OFF，Sleep三条路径。如果Target为Sleep，则进入到Go Sleep阶段，若在该阶段检测到唤醒事件，那么直接添砖至Wakeup Validation阶段。如果Target为OFF或Reset，则需经历Go OFF I与Go OFF II两个阶段，reset则会重新跳转至Startup阶段，而OFF则是直接关闭ECU。

若进入到Wakeup阶段，则需要进行4个阶段的唤醒源验证，主要分为Wakeup I，Wakeup Validation，Wakeup Reaction，Wakeup II阶段；

若进入到Sleep阶段，则可以分为两种Sleep模式：Sleep I 与Sleep II，一般两者选其一。其中Sleep I阶段（Halt），此阶段不运行代码， 等待唤醒事件，然后跳转至Wakeup阶段；其中Sleep II阶段则为Polling阶段，这个阶段则会低功耗运行代码，并且等待唤醒事件，如果存在，则进入到Wakeup阶段。

**Fixed与Flexible模式区别与联系**，从上述EcuM Fixed Mode与Flexible Mode的描述，便可知两者存在着很多的相似点，同时也存在着彼此之间的差异，因此小T我将两者的区别与联系展现如下表所示：



由上分析可知，EcuM Flexible可以兼容Fixed模式，是传统ECU的启动过程的扩展，也可理解Flexible是Fixed模式的更高一层抽象，Fixed则可以称作Flexible模式的一种表现形式。

同时Fixed模式明确了各个阶段的状态及状态切换过程，而Flexible则更为灵活，可以实现多核启动，局部快速启动等特性，为了更好的了解Flexible模式的启动思想，本文将以重点介绍Fixed模式下各状态机的状态机及切换过程，举一反三。

按照EcuM的主体功能，对应的将从以下五个过程来展开讲解EcuM Fixed Mode下的各状态机状态及状态切换过程。

**Startup Sequence : 完成启动过程的初始化；**

**Run Sequence ：正常运行及退出运行状态阶段**

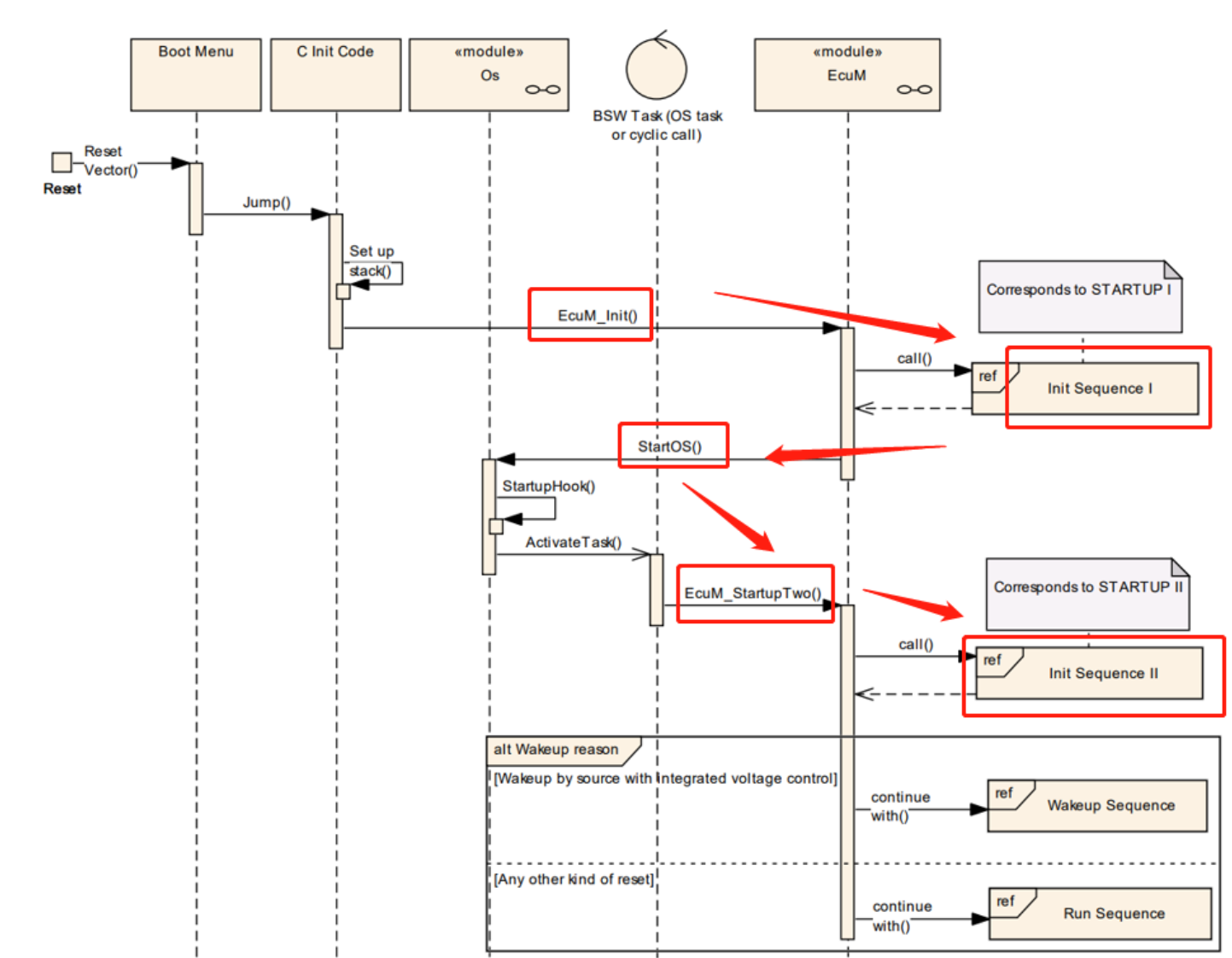
**ShutDown Sequence：shutdown 或Reset ECU的阶段；**

**Sleep Sequence：ECU休眠阶段;**

**Wakeup Sequence: ECU 验证唤醒源阶段；**

Startup Sequence

STARTUP阶段的目的就是初始化基础软件模块，主要可分为两个阶段：启动OS之前的初始化以及启动OS之后的初始化，如下图3所示，为Startup Sequence的顶层设计。



STARTUP I

如上图3所示，通过调用EcuM\_Init函数则进入到STARTUP I阶段，在该阶段主要会调用下列两个Callout函数完成OS启动前的初始化工作；

**EcuM\_AL\_DriverInitZero**：完成无需OS支持的底层硬件驱动的初始化或者其他低水平的初始化（无需postconfig），将这部分驱动的初始化称为Init Block 0；

**EcuM\_AL\_DriverInitOne**：完成无需OS支持的底层硬件驱动的初始化或者其他低水平的初始化，将这部分驱动的初始化称为Init Block 1;

STARTUP II

在STARTUP II阶段则是在start os函数中调用EcuM\_AL\_DriverInitTwo ，随后开启RTE，最后调用函数EcuM\_AL\_DriverInitThree最后初始化那些需要NVM数据的BSW模块。

**EcuM\_AL\_DriverInitTwo** ：需要OS支持但是无需使用NVM的BSW模块初始化，并将此部分驱动的初始化称为Init Block II；

**\*\*EcuM\_AL\_DriverInitThree**：\*\*需要OS支持同时也需要使用NVM的BSW模块初始化，并将此部分驱动的初始化称为Init Block III；

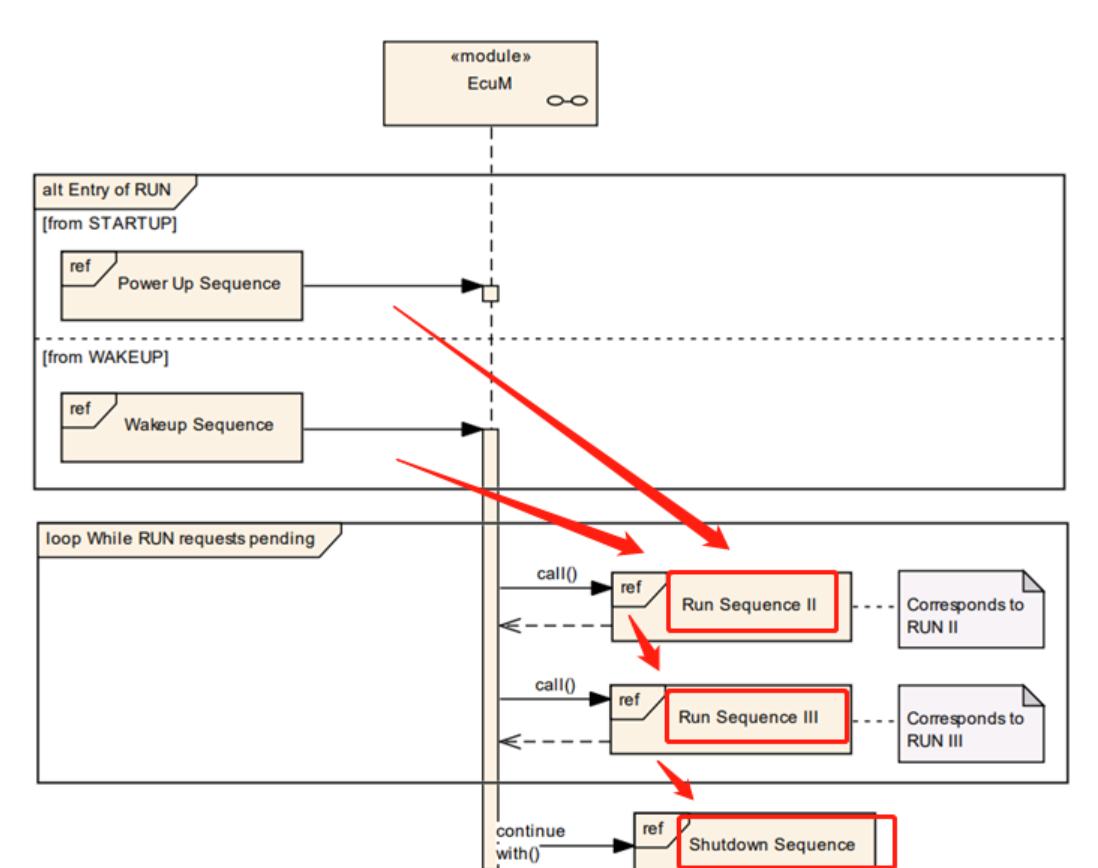
特别需要注意的是，STARTUP 1主要用于为start OS而作的驱动函数初始化，启动时间应当尽可能短，而START UP II尽可能完成所有所需模块的初始化。且中断一般不允许在startup I阶段使用，如果需要使用，也只能使用Category I，不能使用Category II。

为了加深对Startup两个阶段的驱动模块初始化的认识与理解，总结如下表所示：



**RUN Sequence**

RUN阶段可以划分为以下两个阶段，一个是RUN II，表示正常工作状态，另一个是RUN III，表示为进入到ShutDown所作的前提准备，顶层设计如下图所示：



RUN II

在RUN I阶段则表明已完成了所有BSW模块（包括OS及RTE）的初始化，开始运行SW-C程序。在该阶段，将主要完成以下几种操作：

通过调用函数ComM\_CommunicationAllowed来使得相应的通信通道允许通信；

在该阶段，EcuM将允许保持一个最小的运行事件EcuMRunMinimumDuration，以便让SW-C有机会向EcuM模块请求RUN Request；

在该阶段也需要进行休眠总线的唤醒源验证工作；

除非没有通信请求，否则ComM不会释放RUN Request，也就不会退出RUN II阶段；

RUN III

当最后一个Run Request被释放之后，EcuM就会进入到RUN III阶段（即Post RUN 阶段）。在PostRUN主要完成以下几种操作：

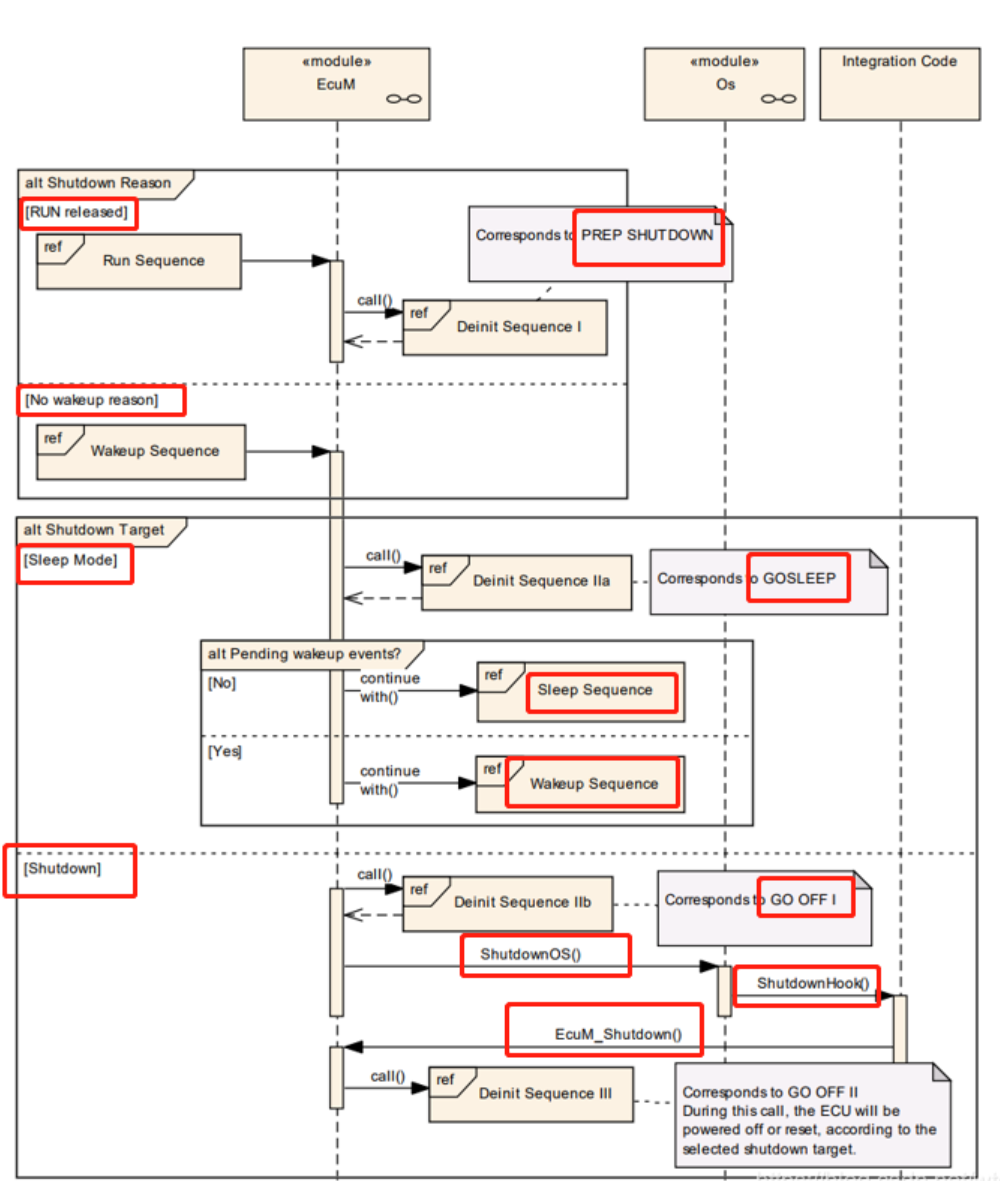
在RUN III阶段，如果Sw-C请求PostRun，那么就会停留在该状态，SW-C可以运行其相应的代码如存储重要的数据等，直至释放PostRun Request；

若在该阶段存在RUN Request，那么就会立刻跳回到RUN II阶段；

若既不存在RUN Request，也不存在PostRun Reqest，那么就会直接进入到ShutDown阶段中的PreShutdown阶段；

ShutDown Sequence

在ShutDown阶段，主要根据ShutDown Target不同而进入不同的状态机处理流程。如下图所示，总体上体现了根据Target不同而做出的不同状态机处理。



从上图可知，不管ShutDown Target是什么，都会经历PreShutdown阶段，进入到该阶段，主要完成以下操作：

De\_Init所有的SW-C，同时保证通信协议栈处于关闭状态。

清除所有的Wakeup Event；

关闭Dem模块；

根据不同的ShutDown目标进入不同的状态（Sleep或者OFF或者Reset）；

ShutDown Target

在ShutDown阶段，ShutDown Target非常重要，因为其决定了ShutDown阶段应当走何种路线。ShutDown Target可分为以下三种：

OFF： CPU掉电；

RESET： 这属于一个暂态，CPU Reset；

Sleep： CPU处于低功耗状态，未掉电；

默认的ShutDown Target可以通过配置得到，当然SW-C可以直接调用函数接口 EcuM\_SelectShutdownTarget来覆盖掉默认的ShutDown Target。

Go Sleep

当ShutDown Target为Sleep时，那么就会进入到Go Sleep阶段，在该阶段主要完成以下操作：

调用NvM\_WriteAll函数完成写操作，同时开启NVM写超时计数器；

调用函数EcuM\_EnableWakeupSources使能Wake up事件接收；

在该阶段，OS并没有关闭，处于正常Running状态；

若此阶段存在Pending Wakeup Event，则直接调用函数NvM\_CancelWriteAll取消写操作，然后直接跳转Wakeup阶段的Wakup Validation子状态；

当Nvm\_WriteAll成功执行完或者写超时，则直接进入到Sleep阶段；

Go OFF I

当ShutDown目标为OFF或者RESET时，则首先进入到该状态。在该阶段，主要完成以下几种操作：

仅设置LIN的通信状态为FALSE；

完成ComM，BswM的Deinit操作；

调用NvM\_WriteAll函数完成写操作，并开启写超时计数器；

等待NvM写成功或者NvM写超时，调用函数ShutdownOS关闭OS；

在ShutDown OS的过程中通过shutdown hook函数调用EcuM\_ShutDown来进入OFF II阶段；

Go OFF II

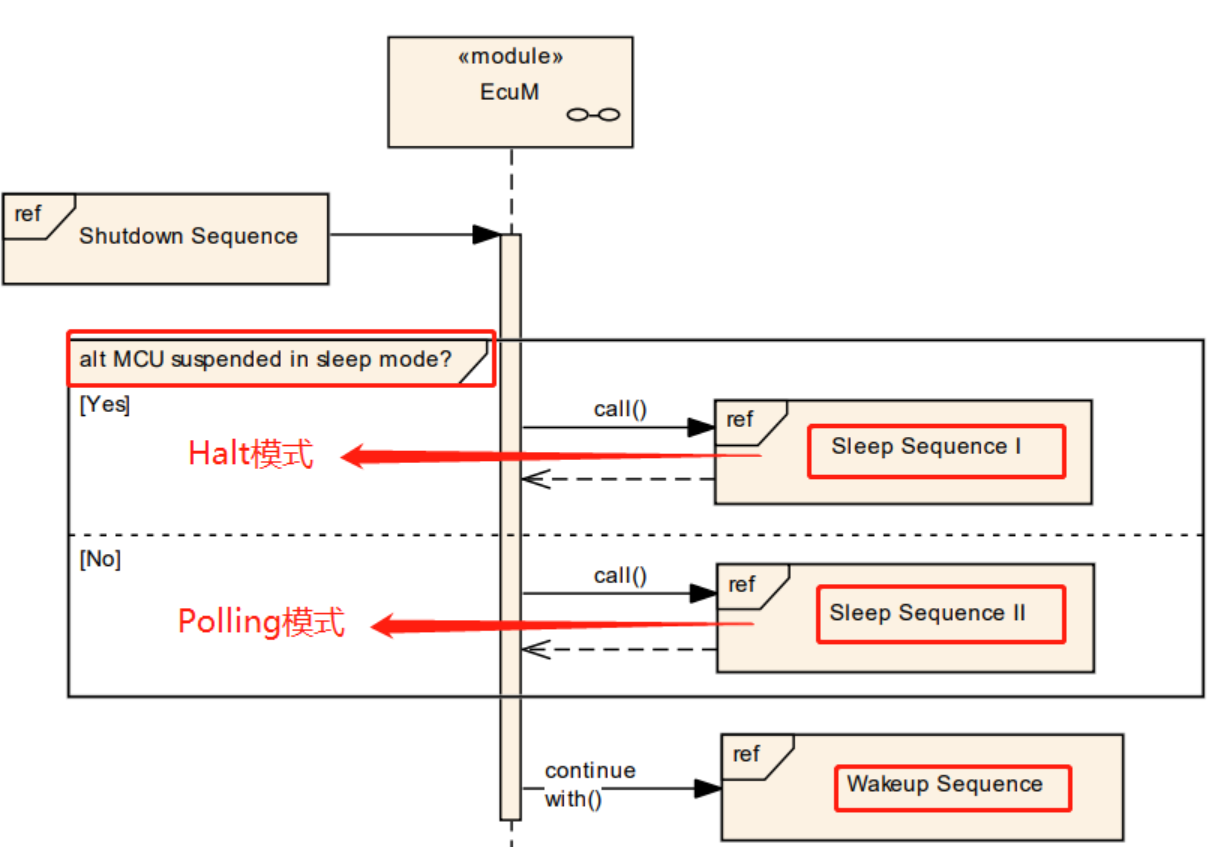
当ShutDown Target为OFF或者RESET时，经过OFF I阶段就会最终调用EcuM\_ShutDown进入到该阶段，在该阶段，主要完成以下几种操作：

如果ShutDown Target是OFF，则调用Callout函数EcuM\_AL\_SwitchOff来直接断掉CPU供电；

如果ShutDown Target是RESET，则调用Callout函数EcuM\_AL\_Reset进而调用MCAL标准函数Mcu\_PerformReset来重启CPU；

Sleep Sequence

当ShutDownTarget为Sleep，经历了Go Sleep阶段后，便会直接进入到Sleep阶段，Sleep阶段的总体流程如下图所示：



如果所有的RUN Request没有被释放，则不会进入到Sleep阶段，也就意味着进入到Sleep阶段了，表示当前已没有RUN Request。在进入Sleep状态之前，EcuM模块应当将所有的通信接口处在Standby状态，且需要使能必要的Wakeup Source。

进入到Sleep模式后，可以选择MCU Halt模式，等待Wakeup Event触发，也可以选择Polling模式，主动查找当前有无唤醒事件，两者根据系统设计选择其中一种即可。

Sleep I

在Sleep I阶段，即Halt模式，在该低功耗模式下，无需运行代码，但需要存在某种CheckSum算法来保证唤醒前后RAM空间的数值不会遭到破坏。

即通过调用EcuM\_GenerateRamHash生成对应的Hash值，接收到唤醒事件后，则调用EcuM\_CheckRamHash来完成前后RAM一致性检查。若一致，则进入到Wakeup阶段，若不一致，则调用Dem模块的Event ID来上报故障并触发重启来保证安全。

Sleep II

在Sleep II阶段，即Polling模式，在该低功耗模式下，会降低系统时钟频率来运行代码，并实时检查有没有相应的唤醒源。通过调用Callout函数EcuM\_SleepActivity以及EcuM\_CheckWakeup来检查是否存在唤醒源。

Wakeup Sequence

如上图2-2所示，无论是在Go Sleep阶段还是Sleep阶段或者是带有电源控制的唤醒阶段，如果监测到Wakeup Event就会进入到该阶段，目前Wakeup Sequence可以分为以下四个基本阶段：

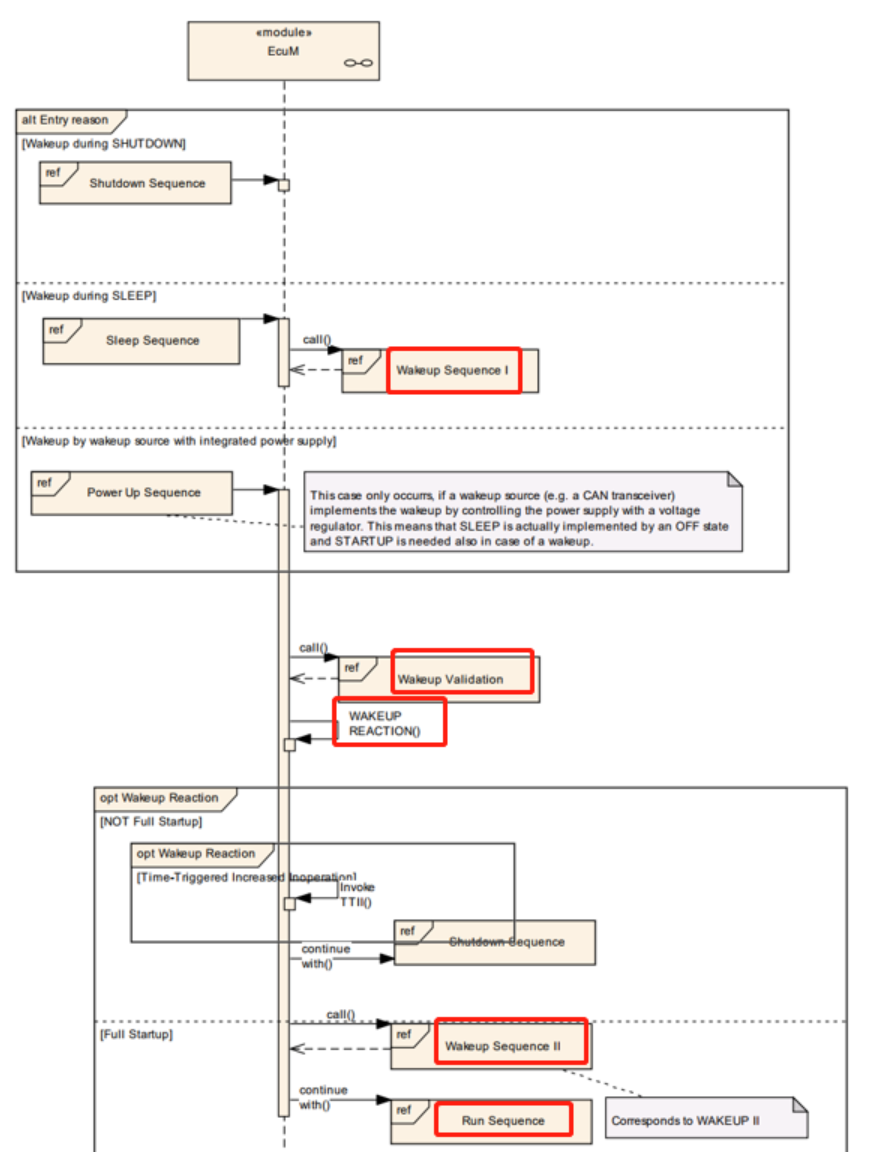
Wakeup One：

Wakeup Validation

Wakeup Reaction：

Wakeup Two：

如下图为Wakeup Sequence的总体流程图：



Wakeup I

当从Sleep状态进入到Wakeup阶段时，首先进入到Wakeup I阶段，在Wakeup I阶段主要完成以下几种操作：

设置MCU模式为Normal Mode；

抑制当前pending的Wakeup Event；

调用函数EcuM\_AL\_DriverRestart重新启动驱动，主要初始化Block I与Block II；

使能Run Reqest以及PostRun Request；

解锁Scheduler并可能重新运行OS；

Wakeup Validation

当从Go Sleep或者通过待电源控制的唤醒条件下启动时，则会进入到该阶段，在该阶段主要会进行以下操作：

获取当前Pending Wakeup Event并调用函数EcuM\_ValidateWakeupEvent开启验证；

如果validate超时，则可以通过调用函数EcuM\_StopWakeupSources停止验证工作；

在该阶段，存在以下5种唤醒源在任何时刻都无需验证：

ECUM\_WKSOURCE\_POWER；

ECUM\_WKSOURCE\_RESET

ECUM\_WKSOURCE\_INTERNAL\_RESET；

ECUM\_WKSOURCE\_INTERNAL\_WDG ；

ECUM\_WKSOURCE\_EXTERNAL\_WDG；

Wakeup Reaction

经过Wakeup Validation阶段后，肯定会进入到该阶段，在该阶段主要会进行以下几个操作：

根据event Validation之后的结果选择进入不同的阶段，一种是验证有效，进入RUN II阶段，另外一种是验证无效，进入Go Sleep阶段；

Wakeup II

当经过Wakeup Reaction之后，如果验证成功就会进入到该阶段，在该阶段主要完成以下几类操作：

如果是从Sleep阶段跳转至该阶段，则首先要调用Dem\_Init函数来完成Dem模块初始化，因为是新一轮operation cycle；

如果是从Startup阶段跳转至该阶段，则可能需要等待NvM readall操作完成；

最后可直接跳转至RUN II阶段直接运行；

**常用函数接口：**

为了更好的使用该模块函数以及遇到问题时方便调试该模块，特将BswM模块中较为重要的常用函数列举如下表所示。

