TriCore™ AURIX™家族系列

TC27xC

TC27xC 启动与初始化

AP32201

应用笔记

V0.1，2014-05

微控器

免责条款

为方便客户浏览，英飞凌以下所提供的将是有关英飞凌产品及服务资料的中文翻译版本。该中文翻译版本仅供参考，并不可作为任何论点之依据。虽然我们尽力提供与英文版本含义一样清楚的中文翻译版本，但因语言翻译和转换过程中的差异，可能存在不尽相同之处。因此，我们同时提供该中文翻译版本的英文版本供您阅读，请参见 www.infineon-ecosystem.org。并且，我们在此提醒客户，针对同样的英飞凌产品及服务，我们提供更加丰富和详细的英文资料可供客户参考使用。请详见 www.infineon.com

客户理解并且同意，英飞凌毋须为任何人士由于其在翻译原来的英文版本成为该等中文翻译版本的过程中可能存在的任何不完整或者不准确而产生的全部或者部分、任何直接或者间接损失或损害负责。英飞凌对于中文翻译版本之完整与正确性不担负任何责任。英文版本与中文翻译版本之间若有任何歧异，以英文版本为准，且仅认可英文版本为正式文件。

您如果使用以下提供的资料，则说明您同意并将遵循上述说明。如果您不同意上述说明，请不要使用本资料。

版本 2014/05

出版发行：
英飞凌科技公司
上海，中国

免责条款

本应用笔记中给出的信息仅作为实现英飞凌器件的建议，不得被视为英飞凌器件的任何特定功能、条件或质量作出的任何说明或保证。此应用笔记的接受者必须在实际应用中判定此种描述的任何功能。英飞凌科技在此否认承担此应用笔记中任何和所有信息相关的任何形式的保证和责任（包括但不限于不侵犯第三方知识产权）。

信息

有关技术、交货条款及条件和价格，请与您最近的 Infineon Technologies 办事处联系。

警告

由于技术要求，组件可能含有危险物质。如需相关型号的信息，请与您最近的 Infineon Technologies 办事处联系。如果可能合理地预期此类组件的故障会导致生命支持器件或系统发生故障或影响该器件或系统的安全性或有效性，则 Infineon Technologies 提供的组件仅可用于获得 Infineon Technologies 明确书面批准的生命支持器件或系统。生命支持器件或系统的目的是植入人体或支持和/或保持并维持和/或保护生命。如果出现故障，则可能危及使用者或他人的人身安全。

文献修订史

日期	版次	修订人	修订内容
2012/01/09	V0.1	Mathews Bejoy	初版

商标：

Infineon®是英飞凌科技公司注册商标。

请留下您的宝贵建议

您是否认为本文档中的任何信息存在错误，含糊不清或遗漏？您的宝贵意见和建议将帮助我们持续不断地改进文档质量。请将您的建议（请注明文档的索引号）发送电子邮件至：

ctdd@infineon.com

目录

1 概述

TC27xC TriCore™ AURIX™ 微控器初始化过程，包括以下步骤：

⇒ 初始化 CPU0 的 C 运行环境（Cstart 程序），AURIX 复位后，CPU0 是默认激活的 CPU。

⇒ 复位测试确定是否需要重新配置时钟。然后执行 RAM 一致性检查，以确保 RAM 测试和初始化的正确完成。

⇒ 之后进行默认驱动程序的初始化，使系统及其外围设备快速进入默认状态。驱动程序进一步的初始化可在 CPU0 上继续进行，也可切换到 CPU1，如果驱动程序初始化过程需要锁步核校验。

⇒ 执行重要安全部件的启动过程的安全测试，应用程序相关软件钩子程序也要进行相应的初始化。

⇒ 然后对普通外设和驱动程序进行初始化，包括初始化时钟和在其它不同的驱动程序中对其余的硬件部分进行初始化。

⇒ 多核系统初始化，包括启动其余 CPU，建立相应 C 运行环境。

⇒ 初始化最后阶段，程序控制权转移到操作系统，程序在多核的同步运行。

另外，本文档还描述了与器件初始化有关的方面：

⇒硬件和固件相关的配置 – 配置硬件引脚，启动模式索引，以及 Flash 中的用户配置模块等。

⇒CPU 和安全看门狗的处理 – 应用笔记 AP3222 对这些内容有进一步描述。

⇒RAM 测试部分 – 在 C 运行环境建立前后，使用存储测试单元进行 RAM 测试。应用笔记 AP32197 将对这些内容进一步描述。

⇒中断和陷阱的设置。

⇒二级启动加载配置和支持，启动加载程序跳转到到应用程序。

⇒ TC27xC 与标定有关的 ED 部分的初始化。

1.1 术语定义及缩写

BMI	Boot Mode Index
ED	仿真器件
HWCFG	硬件配置引脚
HSM	硬件安全模块
ABM	备用引导模式
EVR	嵌入式稳压器。EVR13 生成 1.3V 电压；EVR33 生成 3.3V 电压。
LDO	低压差线性稳压器
SMPS	开关模式电源稳压器拓扑
EOL	行结束编程
CRC	循环冗余校验
UCB	Flash 用户配置模块
MTU	存储器测试单元
MBIST	存储器内部自检
PSPR / DSPR	程序暂存内存/数据暂存内存
SCU	系统控制单元
SSW	启动软件或固件
PSW	程序状态字寄存器
CSA	上下文保存区
BIV	基本中断向量表
BTV	基本陷阱向量表
ECC	纠错码
ISP	中断堆栈指针

热复位	通过 PORST 的复位信号或应用程序或系统复位触发，热复位。不是因为电源故障引起。这样会确保不丢失 RAM 数据并保正数据一致性。
PORST 冷复位	冷上电复位指（1.3V，3.3V 或是 5V）其中一种供电情况下，由电源上升或是电源故障引起的复位。冷复位发生时，不能保证 RAM 存储器数据的一致性。
ESR0 引脚	ESR0 引脚表示外部服务请求引脚 0。ESR0 是默认的硬件复位输出引脚。经过复位后一段可配置的延迟时间，复位输出引脚可能会建立解除。
STADD	STADD 表示执行 ABM/BMI 标头配置的固件后用户代码的启动地址。STADD 代表了 flash 可配置复位向量地址，也是用户代码第一条指令的位置。

1.2 参考文献

本部分列出应用笔记中所有参考文献。

[1] TC27xA microcontroller target specification : tc27x_ts_V2.4_OPEN_MARKET.pdf

[2] Errata sheet TC2D5ED_AA_Errata_Sheet_v1_0

[3] Core specifications:TC_Architecture_volX_TC161_TCS_TC16P_TC16E.pdf

[4] Safety Lib :Aurix-HE_SafeTlib_SAS.pdf

[5] Autosar MCU driver specifications :AUTOSAR_SxS_MCU_Driver.pdf V3.0.0 (Rel 4.0)

[6] AP32197 MTU application note

[7] AP32221 Watchdog application note

2 器件初始化前的硬件和 Flash 配置

本节将讨论 Aurix 启动的前提条件和硬件配置，这些会直接影响到 Aurix 的起动过程和顺序。引脚配置由 HWCFG 引脚确定，而 flash 起动配置，通过编程存放在 ABM/BMI 头和 flash 中的用户配置块（PROCOND）。

2.1.1 硬件配置引脚

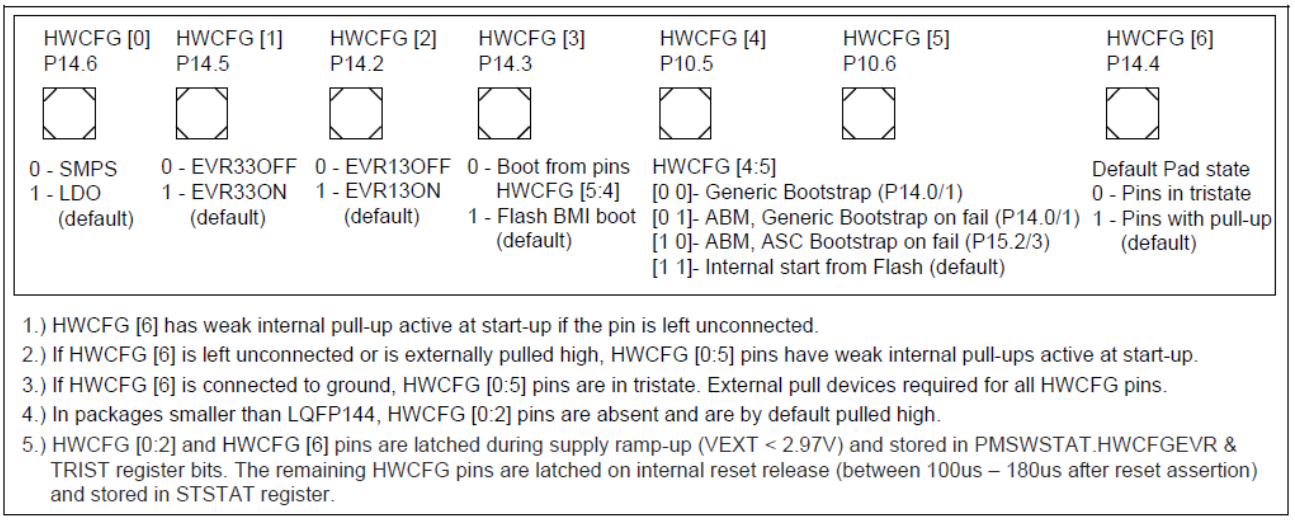


图 1 HWCFG 引脚和功能

1.HWCFG[0:2]引脚，可以配置选择供电模式。通过这些引脚，可以激活内部 EVR13（开关电源模式或线性电源模式）或 EVR33（线性电源模式）或同时激活，可以产生 1.3V 和 3.3V 电源。通过内部或外部上拉器件，确保在电源上电过程中，这些引脚处在正确的电平，避免进入其它的供电模式。引脚的状态被锁存，通过 PMSWSTAT.HWCFGEVR 状态位和 EVRSTAT 寄存器，可以确定电源的供电模式，进而判断是否为期望的模式。

2.HWCFG[3]是 BMI 选择引脚，它决定了起动配置是从硬件配置引脚 HWCFG[4:5]选择，还是从 FLASH 中启动模式索引选择。启动模式选择用 Flash 中的 BMI 的原因主要是可以节省引脚资源，而且可以避免安全漏洞。BMI.PINDIS 位决定是否使能 HWCFG[3:5]引脚用作启动模式选择。

3.只有当 HWCFG[3]配置为由引脚而不是从 FLASH 起动时，HWCFG[4:5]才需要配置基本自举（bootstrap）模式选择。而当用 Flash BMI 作起动选择时，则忽略这些引脚。备用起动模式（ABM），建议使用 P14.0/P14.1 的普通自举模式，因为这是空白 flash 或是启动失败时默认启动加载机制。普通自举指的是 CAN 或 ASC 启动加载程序能自动检测，并起动相应的起动加载过程。

4.HWCFG[6]确定在复位时或复位后，端口引脚是配置为默认的三态，还是带上拉的输入。目前，TC27x A step 没有使用该引脚。HWCFG [6]选择默认的上拉，可以保证所有引脚在悬空时，处于默认的上拉状态。

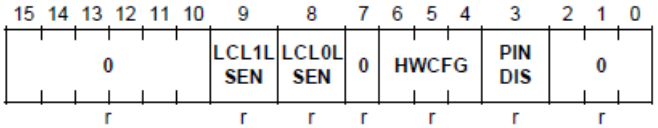
HWCFG 引脚	配置示例
HWCFG [0:2]	依据相应供电模式，与外部上拉或下拉电阻相连接
HWCFG [3]	当使用 FLASH BMI 启动时，产品应用中可以使用该引脚。当 BMI.PINDIS 使能(BMI.PINDIS=1)时，启动时忽略该引脚。
HWCFG [4:5]	当使用 FLASH BMI 索引时，产品应用中可以使用这两个引脚。
HWCFG [6]	与固定外部下拉相联（选择三态为默认模式）
VGATE1P	TC26x & TC24x 中，引脚需要连接至地，用来选择内部通路器件（pass device）。当 EVR13 选择为开关电源模式时，需要连接至 P 沟道 MSOFET。
VGATE1N /P32.0	当 EVR13 选择为开关电源模式时，需要连接至 N 沟道 MOSFET。否则，默认情况下作为端口引脚（P32.0）使用。

2.1.2 Flash 用户配置-启动模式索引和 ABM 标头

当对出厂空片初次编程时，完整的 ABM 头包括 BMI（启动模式索引）将会被写入 Flash。

启动模式索引有两字节的数据，包含启动模式等信息，需要在下线编程时写入。BMI 头由 9 个字段组成，并存储在 Flash 中最多 4 个不同地址的位置，而且至少有一个位置写入了 BMI 头。BMI 头存放在多个地址位置，可以用作 BMI 更新时的数据冗余，保证某一个或多个 BMI 头损坏时，Aurix 仍能够使用 BMI 启动。

注意：当所有 BMI 头测试或代码检查失败时，将使用 P14.0/P14.1 进入普通自举模式。因此建议在 EOL 基本自举加载程序或 ECU 的现场编程中，使用的这些引脚。

<table>
<tr><th>偏移地址</th><th>大小（字节）</th><th>ABM 字段名称</th><th colspan="3">描述</th></tr>
<tr><td>00_H</td><td>4</td><td>STADABM</td><td colspan="3">用户程序起始地址</td></tr>
<tr><td rowspan="5">04_H</td><td rowspan="5">2</td><td rowspan="5">BMI</td><td colspan="3">启动模式索引 (BMI)
 
</td></tr>
<tr><th>字段</th><th>位</th><th>描述</th></tr>
<tr><td>PINDIS</td><td>3</td><td>配置引脚选择模式
0 使能 HWCFG 引脚选择模式
1 禁用 HWCFG 引脚选择模式</td></tr>
<tr><td>HWCFG</td><td>[6:4]</td><td>启动模式选择
111_B 从 Flash 内部启动
110_B 备用引导模式(ABM)
100_B 普通自举加载
011_B ASC 自举加载</td></tr>
<tr><td>LCL0LS EN</td><td>8</td><td>CPU0 锁步比较逻辑控制
0 禁用 CPU0 锁步
1 使能 CPU0 锁步</td></tr>
<tr><td>LCL1LS EN</td><td>9</td><td>CPU1 锁步比较逻辑控制
0 禁用 CPU1 锁步
1 使能 CPU1 锁步</td></tr>
<tr><td>06_H</td><td>2</td><td>BMHDID</td><td colspan="3">启动模式头 ID(确认码)=$B359_H$</td></tr>
<tr><td>08_H</td><td>4</td><td>ChkStart</td><td colspan="3">存储器范围-开始地址</td></tr>
<tr><td>$0C_H$</td><td>4</td><td>ChkEnd</td><td colspan="3">存储器范围-结束地址</td></tr>
<tr><td>10_H</td><td>4</td><td>CRCrange</td><td colspan="3">存储器范围校验值</td></tr>
<tr><td>14_H</td><td>4</td><td>CRCrange</td><td colspan="3">存储器范围校验反值</td></tr>
<tr><td>18_H</td><td>4</td><td>CRChead</td><td colspan="3">ABM 头(偏移 00_H..17_H)校验值</td></tr>
<tr><td>$1C_H$</td><td>4</td><td>CRChead</td><td colspan="3">ABM 头校验反值</td></tr>
</table>

1. n（n = 0,1,2,3）个 BMI 头的 FLASH 地址见下表。BMI 是通过非缓存段（non cached segment）进行寻址的， 陷阱（TRAP）向量表跟在 BMI 后面。所以，举例说如果 BMI/ABM 在 A000 0000h，检查相应的 BMI/ABM 的内容后，之后缓存的段地址 A0000 0020h 可以作为跳转的向量表。

BMI 标头	起始地址	结束地址
BMHD0	A000 0000_H	A000 001F $_H$
BMHD1	A002 0000 $_H$	A002 001F $_H$
BMHD2	A000 FFE0 $_H$	A000 FFFF $_H$
BMHD3	A001 FFE0 $_H$	A001 FFFF $_H$

2. 用户代码的入口地址存储在 BMHD[n].STADABM 字段。固件程序运行结束前，CPU0 从该地址进入用户程序，所以该地址成为入口点或复位向量地址。

3.BMI 索引也被保存，用来选择相应的自举模式。

- BMHD[n].BMI.PINDIS 屏蔽了 HWCFG [3:5]引脚功能。如果该位置位，那么启动配置从 Flash 中的 BMI.HWCFG 索引选择，而不再从引脚选择。
- BMHD[n].BMI.HWCFG 选择某个启动模式。

4.启动模式头 ID 编程为 BMHD[n].BMHDID = B359H。

5.如果选择 ABM 模式，将计算出从 BMHD[n].ChkStart 到 BMHD[n].ChkEnd 存储范围内的数据的 CRC 校验和，并与 BMHD[n].CRCrange 存储的值进行比较。而且，将计算结果取反，与 BMHD[n].$\overline{\text{CRCrange}}$(偏移 14H)进行比较。因此，如果选择这种模式， 需要先计算出 ChkStart, ChkEnd，CRCrange 和/$\overline{\text{CRCrange}}$ 并存储在 ABM 头内。

6.如果选择 ABM 模式，需要计算出 ABM 头（00H...17H）的前 24 个字节数据的 CRC 校验和，与 BMHD[n].$\overline{\text{CRChead}}$ 存储的值进行比较。另外，将计算结果取反，与 BMHD[n].CRChead(偏移 1CH)比较。因此，如果选择这种模式，需要根据 ABM 头（00H...17H）的内容，计算出 BMHD[n].CRChead 和 BMHD[n].$\overline{\text{CRChead}}$，并存储在 BMI 头内。

BMI 配置	配置示例
BMHD[n]地址	BMI 在写入到其它位置时，都会写入一次到 A000 0000_H - A000 $001C_H$。需要注意的是，这些位置不用时，要通过链接器将这些存储位置排除掉， 使它们不会被应用程序代码使用。
STADABM	A000 0020_H
BMI	0268_H
- BMI.PINDIS	禁用引脚选择启动模式。BMI.PINDIS = 1
- BMI.HWCFG	= 110 $_B$ - 从 Flash 内部启动

– BMI.LCL0LSEN	= 0 – CPU0 锁步禁用。(仅 TC27xB step 可用)
– BMI.LCL1LSEN	= 1 – CPU1 锁步使能。(仅 TC27xB step 可用)
BMHDID	$B359_H$
ChkStart	$0000\ 0000_H$
ChkEnd	$0000\ 0000_H$
CRCrange	$0000\ 0000_H$
$\overline{\text{CRCrange}}$	$0000\ 0000_H$
CRChead	t.b.d.
$\overline{\text{CRChead}}$	t.b.d.

根据 IEEE 802.3 标准确定的标准 CRC32 多项式，可计算出范围和头部数据的 CRC 校验和。

$$CRC\ 32 = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + 1$$

```
void _RESET (void) // ABM header located and programmed at A000 0000h.
{                                   // Code only intended for development phase
        __asm (".word 0x80000020"); // STADBM first user code at 0x8000 0020h
        __asm (".word 0xb3590268"); // BMI = 0268h BMHDID = B359h
        __asm (".word 0x00000000"); // ChkStart
        __asm (".word 0x00000000"); // ChkEnd
        __asm (".word 0x00000000"); // CRCrange
        __asm (".word 0x00000000"); // !CRCrange
        __asm (".word 0xtbd");              // CRChead
        __asm (".word 0xtbd");              // !CRChead
        /* After firmware evaluates the ABM, jump to STADABM address follows.*/
}
```

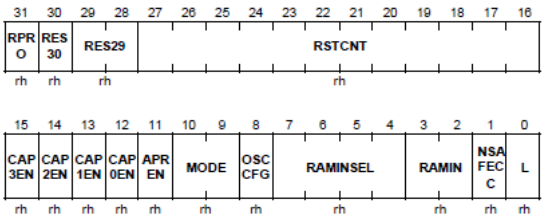
图 2 BMI 头

2.1.1 Flash (PROCOND)的用户配置和固件初始化

根据 Flash 用户配置块（DF0/UCB1）中的预存设置，固件程序会执行相应的 RAM 区初始化，振荡器电路的配置和 ESR0 硬件复位输出。

由电源产生的冷上电复位后，所有的 RAM 区会完成基本的初始化。当 PROST 引脚复位，或应用复位，或系统复位请求建立时，能要保证 RAM 内容本身的一致性。空闲-请求-确认序列能够保证微控制器在收到热复位请求时，在完成所有内存访问时才执行复位，防止内存崩溃。冷上电复位后，通过程序写入 PROCOND.RAMIN = x0H,可以选择性地初始化内存区。这样，在冷复位后，固件程序将内存区初始化为 0，而热复位事件发生后，内存区数据保持不变。

通过 PROCOND.RAMINSEL 位，可以选择某些内存区进行相应的初始化。每个 CPU 和 LMU 中的内存区会被初始化，而且相应用于检测内存故障的 ECC 值也被正确初始化。需要注意的是，出于安全考虑，在每次 PROCOND. RAMIN 或 RAMINSEL 位置位激活 MTU 时，都会执行内存的初始化。而且，在 MTU 的范围（RANGE）功能用来选择需要初始化和测试的内存范围，建议通过 PROCOND.RAMIN 或 RAMINSEL 寄存器位可以禁用内存的初始化。

数据 FLASH 地址 DF0 / UCB1	内容	描述
AF10 0400$_H$	PROCOND	DFlash 扇区，EEPROMs，振荡器配置以及内存初始化使能的读写保护 
AF10 0410$_H$	PROCOND	PROCOND 内容备份
AF10 0420$_H$	PW0 - PW3	256 位密码，从最低位双字到最高位双字（共 4 个双字）
AF10 0440$_H$	PW0 - PW3	256 位密码备份。
AF10 0470$_H$	UCB 状态 确认或是解锁	解锁：4321 1234 $_H$ 接着是 0000 0000 $_H$ 确认：57B5 327F $_H$
AF10 0478$_H$	UCB 状态 确认或是解锁	解锁：4321 1234 $_H$ 接着是 0000 0000 $_H$ 确认：57B5 327F $_H$

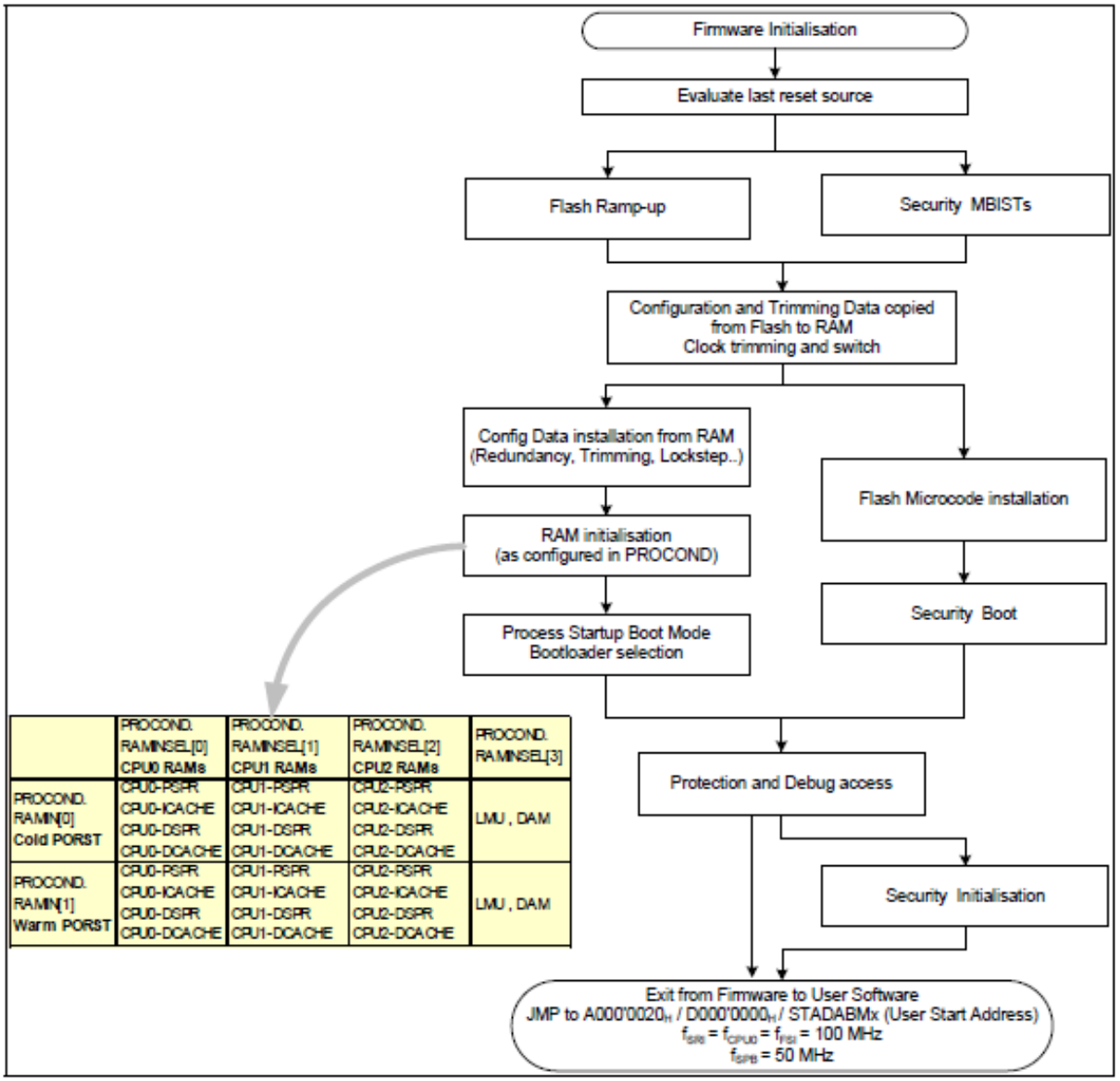


图 3 通用固件程序流程及内存初始化

PROCOND
DFlash 保护配置 （1030_H） 复位值：$0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RPRO	RES30	RES29		RSTCNT											
rh	rh	rh		rh											

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAP3EN	CAP2EN	CAP1EN	CAP0EN	APREN	MODE		OSCCFG	RAMINSEL				RAMIN		NSAFECC	L
rh	rh	rh	rh	rh	rh		rh	rh				rh		rh	rh

字段	位	类型	描述
RAMIN	[3:2]	rh	**SSW 控制 RAM 初始化** 这些位决定是否初始化字段 RAMINSEL 所选择的 RAM。 00 Init_All 冷上电复位和热上电复位后执行 RAM 初始化。 01 Init_Warm 热上电复位后初始化 RAM，冷上电复位不执行初始化（不建议使用）。 10 Init_Cold 冷上电复位后初始化 RAM。 11 No_Init 不执行 RAM 初始化。
RAMINSEL	[7:4]	rh	**RAMINSEL RAM 初始化** 当通过 RAMIN 配置 RAM 初始化时，这些位选择初始化的存储器。 通过每个位置“0”，选择用于初始化相应的存储器。 xxx0 选择用于初始化的 CPU0 的 PSPR, DSPR 和 ICACHE。 xx0x 选择用于初始化的 CPU1 的 PSPR, DSPR 和 ICACHE。 x0xx 选择用于初始化的 CPU2 的 PSPR, DSPR, ICACHE 和 DCACHE。 0xxx 选择用于初始化的 LMU RAM。
OSCCFG	8	rh	**SSW 配置 OSC** 这个位表示是否通过 SCU_OSCCON 的 SSW 安装振荡器配置(字段：CAPxEN, APREN, MODE)。CAPxEN, APREN, MODE) 0 SSW 复位数据到 SCU_OSCCON（忽略 PROCOND 的位 17-23）。 1 通过这个数据 SSW 配置振荡器。
MODE	[10:9]	rh	**OSC 模式** 当通过 OSCCFG 使能时，把这个字段复制到 SCU_OSCCON.MODE。
APREN	11	rh	**OSC 振幅调节使能** 当通过 OSCCFG 使能时，把这个字段复制到 SCU_OSCCON.APREN。
CAPxEN (x=0-3)	12+x	rh	**OSC 电容 x 使能 (x=0-3)** 当通过 OSCCFG 使能时，把这些字段复制到 SCU_OSCCON.CAPxEN。
RSTCNT	[27:16]	rh	**ESR0 延迟** 用于配置 ESR0 延迟。通过 SSW 评测。 000 ESR0 立即释放 001 - FFE ESR0 建立解除延迟=RSTCNT x 10us FFF 通过设置 ESR0CFG.ARI 位，用户软件建立解除 ESR0

PROCOND 配置	配置示例
PROCOND	0000 000FC $_H$
• RAMIN	固件未初始化 11 $_B$ .RAM。 （利用 MTU，有选择地测试和初始化 RAM）
• RAMINSEL	固件未初始化 1111 $_B$. RAM。
• OSCCFG	0 $_B$
• MODE	0 $_B$
• APREN	0 $_B$
• CAP0EN	0 $_B$
• CAP1EN	0 $_B$
• CAP2EN	0 $_B$
• CAP3EN	0 $_B$
• ESR0CNT	000 $_H$. 不会发生 ESR0 建立解除延迟。

3 TC2xA 初始化顺序

本节阐述了 TC27xC TriCore® 微控器初始化过程的主要步骤。如图 4 所示，主要的初始化过程在 CPU0 上执行，或如果在初始化阶段要求锁步核校验的话，也可会切换至 CPU1，如图 5 所示。对于安全应用，可能还需要与 HSM 模块交互操作。

电源供电，电源故障，PROST 引脚复位，系统复位或是应用复位都有可能产生复位，进入启动代码开始执行。基于这些复位事件和单片机的状态，需要执行不同的初始化序列。

1. 首先，初始化 C 运行环境，主要包括了堆栈，上下文存储区以及 CPU0 缓存的测试和初始化。如第 3.1 节所述，进一步初始化一些关键寄存器，如向量表基地址和编译器指定寄存器。
2. 第二步，确定已发生的复位事件类型，并测试是否有电源故障或内存内容是否一致，测试结果确定之后的初始化需要执行的步骤。如果是应用复位，并且内存内容没有被破坏，则不需要初始化内存，时钟和 flash 模块，从而进入快速启动的过程。第 3.2 节对该过程有更详细的描述。
3. 第三步，初始化各种驱动和外设为缺省的状态。例如，根据自举加载程序和应用程序的需要，端口引脚初始化为所需要的初始电气状态，到这会决定由哪个 CPU 继续执行进一步初始化。第 3.3 节对该过程有更详细的描述。
4. 应用程序调用的钩子程序可以集成启动时的安全测试，内存区的测试和初始化。根据配置，执行 CPU 测试，RAM 测试，SMU 警告检验和其他重要的安全启动测试。另外，这里还可加入并调用 SafeTLib 程序。第 3.4 节描述了这部分内容。
5. 完成安全测试后，应用的钩子程序进行应用需要的硬件模块驱动的初始化。接下来执行的步骤，包括调用相应的驱动初始化程序（第 3.5 节所述该内容），时钟初始化，配置 flash，驱动见共享资源的初始化。执行时钟初始化，flash 配置和驱动程序间共享资源的初始化，如不同驱动间共用的 GTM 的初始化程序。
6. 在各驱动程序初始化完成后，需要一个应用的钩子程序来配置操作系统，及启动另外的处理器（CPUs）。然后调用安全检查程序，检查外设的配置。接下来执行的步骤，根据配置启动其它处理器，初始化各处理器相应的 C 运行时态。所以需要另一个钩子程序用于相应处理器的配置，第 3.6 节将进一步描述。
7. 最后，把控制权转到操作系统，对于多核 OS 系统来说，在完成内核同步后，各处理器上都会调用 StartCore()和 Start OS()函数。如果使用了 AUTOSAR 基础软件层，这一步会调用 Ecum_Init()函数，并促发 OS 的初始化，第 3.7 节将进一步描述。

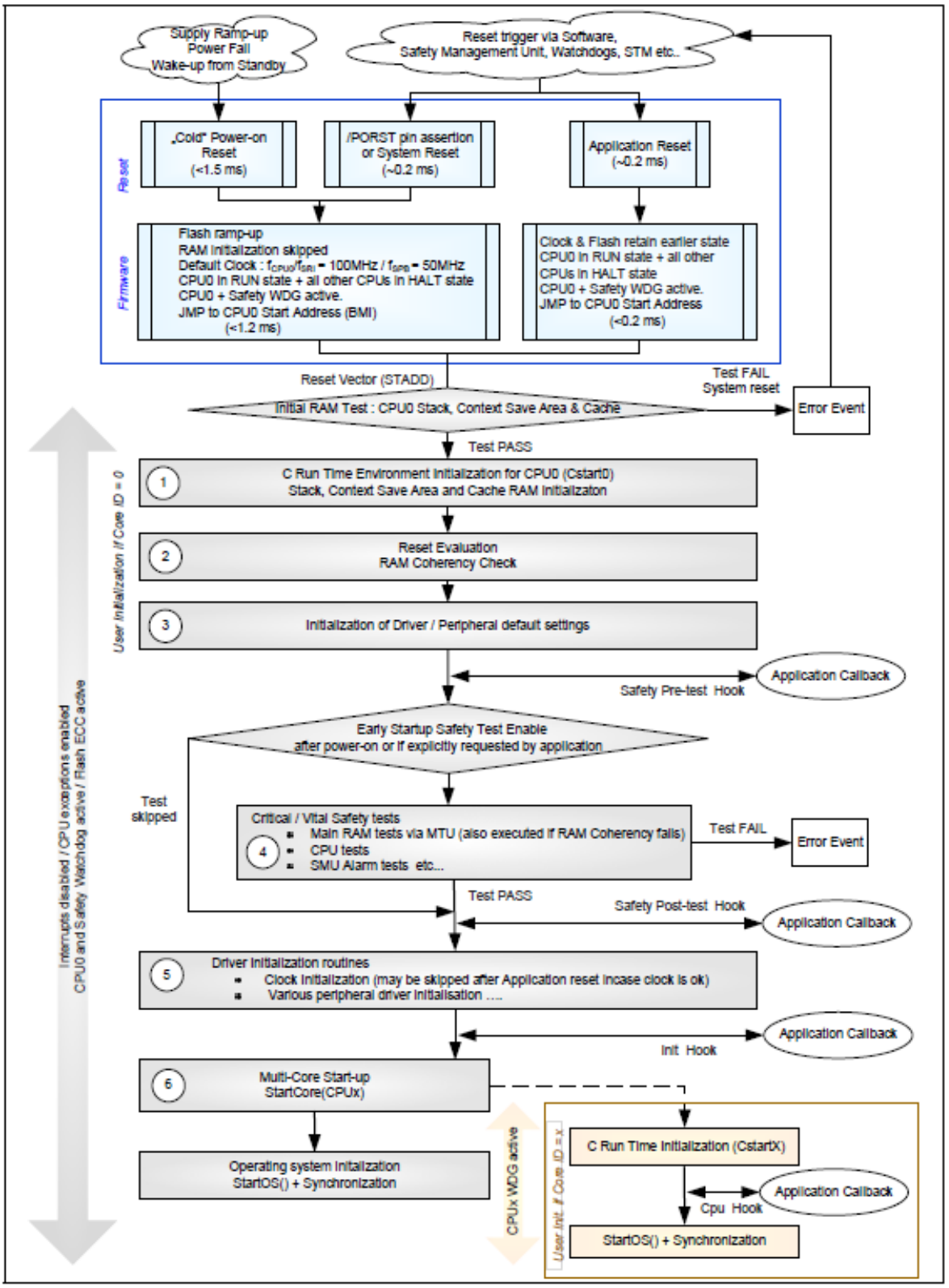


图 4　TC27xA 初始化（CPU0 内核）

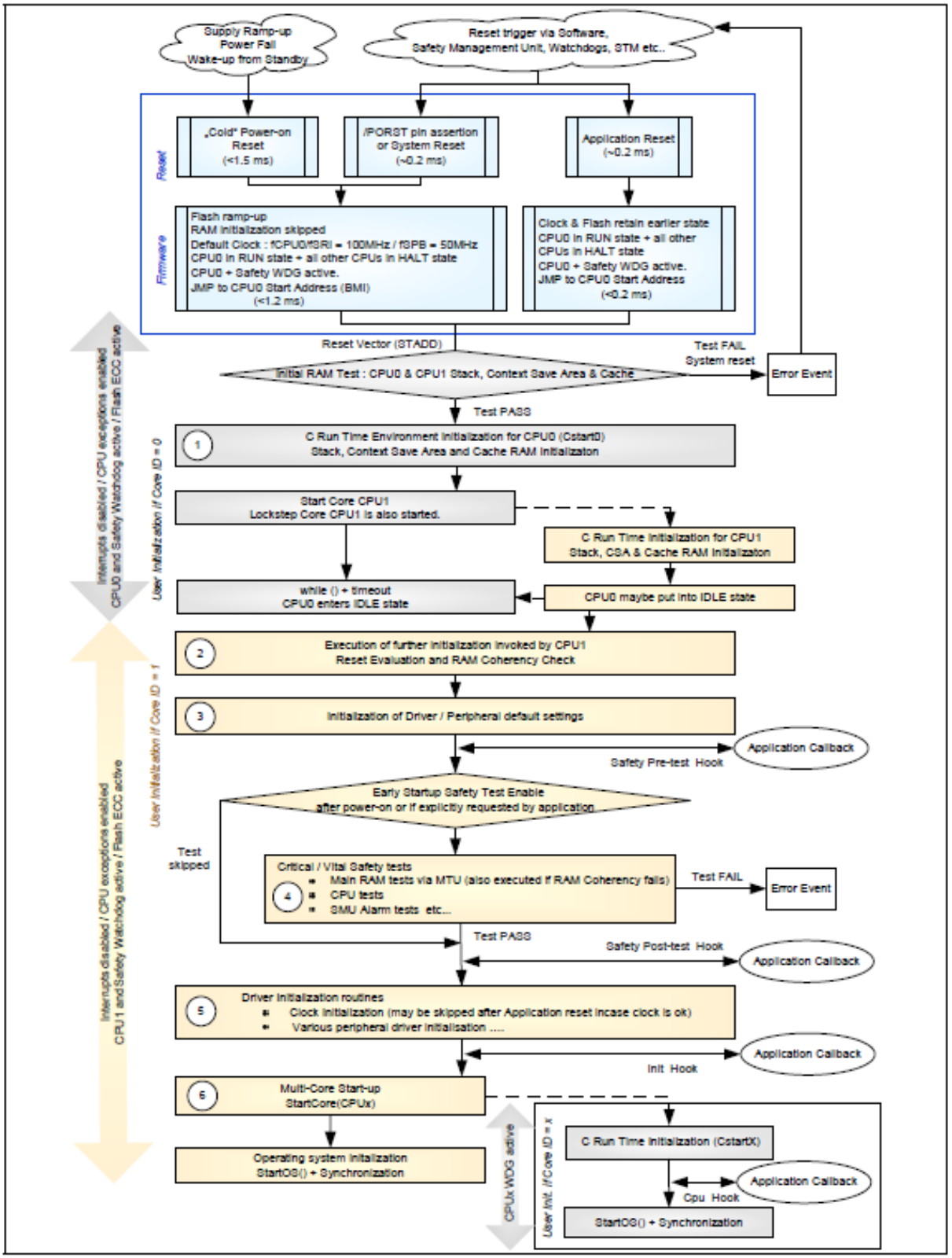


图 5　TC27xA 初始化（CPU1 锁步核）

3.1 C 运行环境初始化

在执行基于 C 语言的函数前，需要初始化和建立基本的 C 运行环境。从复位向量（STADD）开始执行的启动函数 CStart0，封装了初始化 CPU 寄存器和建立程序运行的基础架构。这部分操作不会使用函数调用，全局变量，上下文或堆栈，并且中断产生应该关闭，而陷阱和 Flash ECC 是有效的。每个核应有各自的 CStart 初始化程序。

1. 这部分启动代码将正确初始化用户堆栈指针及程序状态字（PSW）寄存器。A10 寄存器用作堆栈指针，户堆栈指针基地址和堆栈大小，可通过链接器/定位器的配置参数来设置。启动代码应该测试和初始化用于堆栈的存储区，对齐堆栈边界。
2. 启动代码测试和初始化用于保存上下文的存储区，寄存器 PSW(CDC)中指示连续上下文切换的调用深度的内容被重置。通过寄存器 PCXI,LCX 和 FCX，初始化 CSA 链表。通过链接器/定位器的配置参数，设置各核的 CSA 区。
3. 地址寄存器 A0,A1,A8 和 A9 是系统全局寄存器，也要初始化。通常，A0 和 A1 预留给编译器，而 A8 和 A9 留给 OS 或应用程序。A0 用作小数据区（small data section）的基指针，通过基地址 + 偏移地址可寻址访问该区的全局数据元素。A1 用作数据区（literal data section）的基指针。
4. 启动代码初始化中断和陷阱向量表的基地址，也就是各内核的 BIV 和 BTV 寄存器，通过链接器/定位器的配置参数可修改这些设置。
5. 如要支持中断嵌套，启动代码将初始化中断堆栈指针寄存器 ISP。通过链接器/定位器的配置参数，可设置中断堆栈基地址和大小。需要注意的是，BTV，BIV 和 ISP 是带 EndInit 保护的寄存器，可放在一起设置。
6. 测试，初始化和使能数据缓存或程序缓存。根据配置参数，初始化寄存器 PCON 和 DCON 使能数据缓存或程序缓存，用作缓存的内存大小也在这里配置。在初始化之前，先验证缓存区。为了节省时间，从开始时先测试并激活销量缓存。使能缓存后，可加快从 Flash 取指的速度。
7. 启动代码测试并初始化少量内存，可保证正确执行驱动程序服务的起动和调用这些服务程序。
8. 启动代码应保证了内部 CPU0 和安全看门狗一直能被服务，直到看门狗驱动初始化完成。为了避免频繁的看门狗服务，在启动时降低看门狗定时器的速度，加大服务时间。CPU1 和 CPU2 的看门狗，如果有的话，在复位后仍处于禁用状态。然而，在初始安全测试和初始化阶段，也需要服务看门狗，这取决于测试耗费的时间。
9. 清除全局存储区，包括 C 变量，阵列（.bss，.sbss）。一般的分方法是，微控器起动时，将这些未初始化的区域的每个字节都置零，通过 MTU MBIST 程序可以帮助完成这些操作。

10. C init 函数:如有需要初始化某些变量，需要将变量初始值从 Flash 复制到变量的内存区，如变量数据段，或函数内的静态值。链接器/定位器的设置，可以保证这些符号被正确地重定位到内存的地址之后将执行复制 flash 内容，初始化应用程序相关内存。
11. 初始化结束前， 激活 CPU 0 级到 7 级 TIN 陷阱。MMU，内部保护，指令错误，上下文管理，系统总线和外设错误，断言，系统调用和 NMI 陷阱也被激活。在对应陷阱激活之前，清除陷阱标志位。根据实际应用的需要，中断一直保持关闭，到调用 OS 才打开。

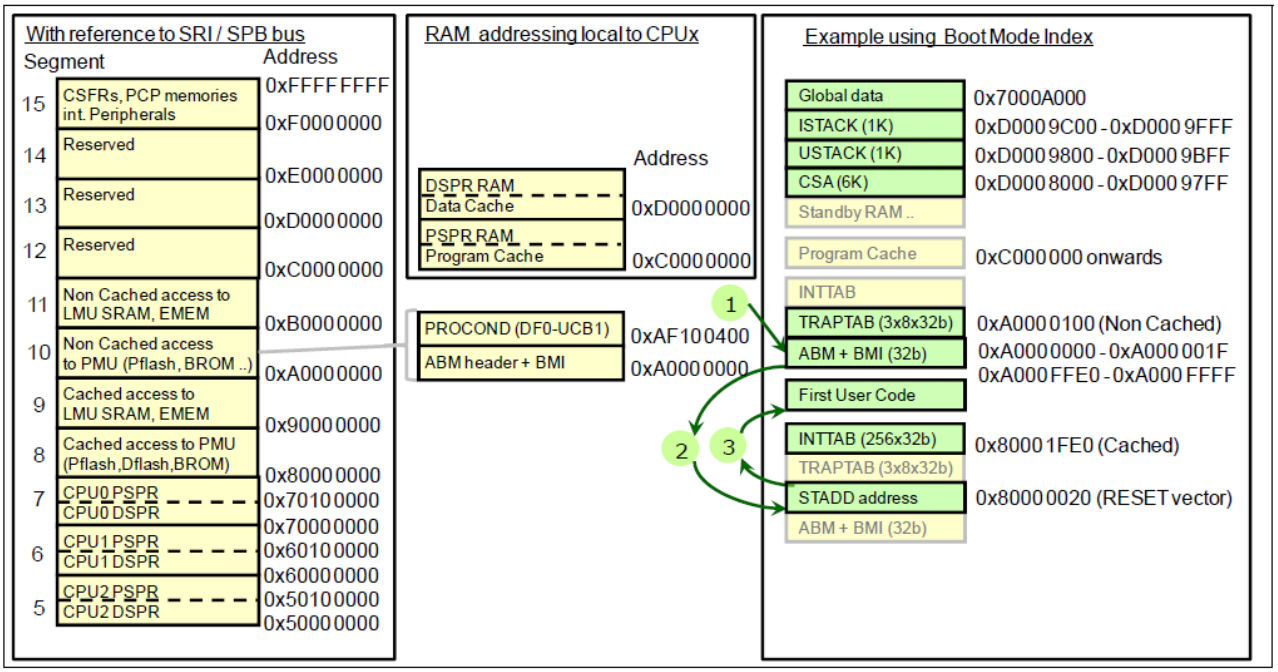


图 6 存储器映射和起动流程

表 1 链接器/定位器符合列表

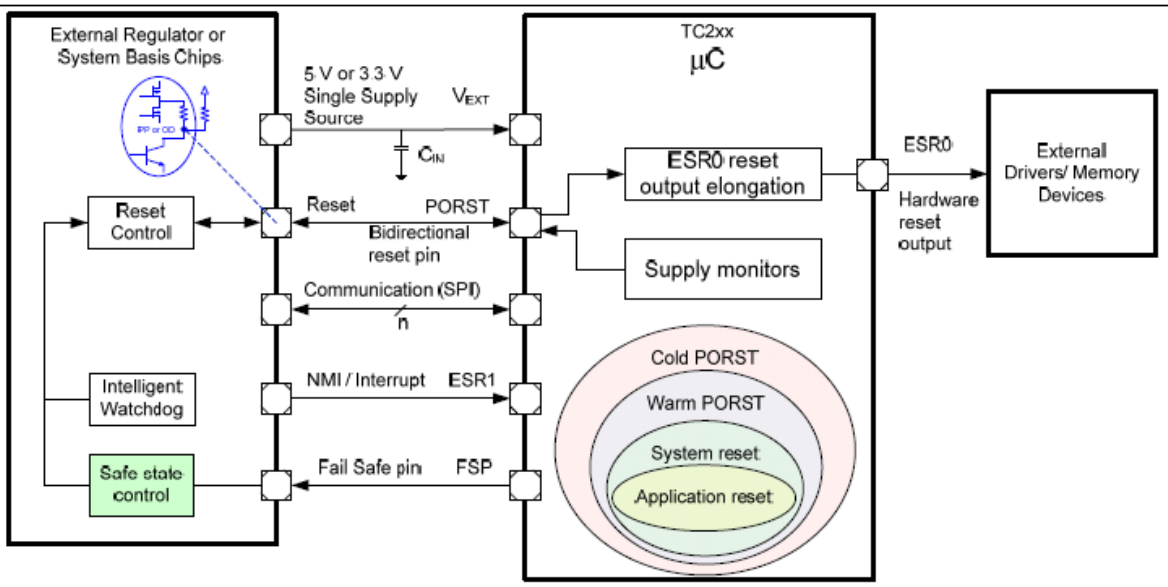
____ISTACK<core number> ____ISTACK<core number>_END ____USTACK<core number> ____USTACK<core number>_END	中断堆栈起始和结束地址 用户堆栈起始和结束地址
____SDATA1_<core number> ____SDATA2_<core number> ____SDATA3_<core number> ____SDATA4_<core number>	A0 可寻址区地址 A1 可寻址区地址 A8 可寻址区地址 A9 可寻址区地址
____CSA<core number> ____CSA<core number>_END	上下文存储区起始与结束
____INTTAB<core number> ____TRAPTAB<core number>	中断向量表指针(BIV) 陷阱向量表指针(BTV)

3.1.1 C 运行环境初始化的内存测试

使用 MTU 测试程序，对基本 CSA,堆栈和缓存等内存相应区域进行测试。需注意的是，这部分程序不能调用函数，测试时也不能使用堆栈或是缓存。本文示例中，会用到 March 测试，Checkerboard 测试和非破坏性反向测试等算法，来测试这些区域。关于 MTU 模块的使用，如何进行存储器测试，应用笔记 AP32197 中有相应的解释。

3.2 复位和内存一致性测试

复位测试能够区分不同类型的复位或电源错误事件，这些复位可以将内部模块，寄存器和端口引脚置为复位状态。基于发生的复位类型，需要执行不同的初始化动作。一个驱动循环仅执行一次安全初始化，因此在低级别复位时，会跳过此过程。同样，应用复位时通常会跳过时钟和 Flash 的初始化。需要有相应的方法，能够识别复位源，并报告给其它查询的软件。



Reset Type	Reset Trigger	Modules affected by any reset	Modules affected additionally by System reset	Modules affected additionally by warm PORST	Modules affected additionally by Cold Power-on
Application Reset	▪ESR0/ESR1 ▪SMU ▪STMx ▪Software reset ▪Tuning protection	▪All CPUs ▪All Peripherals ▪Port pins in reset state ▪Parts of SCU ▪RAMs - Dcache invalid - Pcache invalid			
System Reset	▪ESR0/ESR1 ▪SMU ▪STMx ▪Watchdog (SMU) ▪Software reset		▪Flash memory ▪XTAL/ Osc./ PLL ▪ESRx pins		
Warm Power-on Reset	▪PORST pad asserted			▪JTAG interface ▪OCDS / MCDS ▪SMU-FSP pin	
Cold Power-on Reset	▪Startup ▪Ext. supply (SWD) < 3.0V ▪EVR33 supply < 3.0V ▪EVR13 supply < 1.17V				▪EVR ▪Internal clocks ▪RAMs -DSPRs/ PSPRs -LMU/BMU

A higher reset encapsulates a lower reset
A higher reset resets all the modules reset by a lower reset

SW Module reset	▪<Module>_KRST0.RST ▪<Module>_KRST1.RST	▪DMA Channel, QSPI, CAN, ASCLIN, Flexray, Ethernet, MSC, HSSL, GTM, CCU6, GPT12, ADC, SENT, ADC …
Debug Reset	▪OCDS request trigger ▪JTAG reset	▪OCDS + MCDS reset , All CPUs and peripherals (except SCU) are put into reset.

图 7 复位类型，触发和效果

1. 启动阶段，通过某种方法可以查询复位状态，从而获取当前复位事件的复位类型和触发源，如下图所示。测试包括查询一下状态信息。
   - 电源状态：包括检查寄存器 PMSWSTAT 的 HWCFG 状态，检查寄存器 EVRSTAT 获得外部通过器件（pass device）的状态，确保供电模式和预期的一致。退出待机模式时，通过 PMSWSTAT 可获得待机内存状态。在正常使用时，待机内存状态也可以通过查询寄存器 PMSWSTAT 获得。
   - 复位测试：从寄存器 RSTSTAT,EVRSTAT 和 PMSWSTAT，可以推断出复位产生的原因。冷复位标志位（SWD, EVR13 和 EVR33）会所存到后寄存器 RSTSTAT，这些标志位通过寄存器 RSTCON 清零，且只有在冷复位时，才要求清零。热复位时，复位状态值锁存到寄存器 RSTSTAT，并保持到下个复位发生。用寄存器 RSTCON2.USRDATA 可以判断软件复位类型（SW_RES1 和 SW_RES2），这个寄存器的值在两个安全节拍内才能预测出，再要一个时钟周期写入。
   - 内存一致性检查：检查 CPUs 和内存间的空闲-请求-确认序列是否成功完成，早期复位阶段的内存访问是否正确结束，通过检查 RSTCON2.FRTO, FLSS 和 CSS 就可以做出判断。
2. 起动驱动程序需要提供某种软件的方法，能够触发系统复位或应用复位。当软件检测到某个特定的未知系统状态，该方法强制微控器硬件进行控制下的初始化。这些可能由某些严重错误触发，如发生不可恢复的 µC 陷阱（如 ECC 陷阱），或是软件状态机进入未定义状态（如因为内存的位错误），系统不再有其它动作。该复位方法通过保存在寄存器 USRDATA 的值传递参数，这个值在复位后会可以被读出来。如果需要产生某种复位，通过设置寄存器 RSTCON 中的复位类型，然后置寄存器 SWRSTCON 的 SWRSTREQ 位发出相应的复位请求。
3. 应用复位时，不要求重新初始化时钟和 flash 参数，尽管建议检查时钟配置。驱动程序初始化时，会做时钟的配置。应用复位可以进行快速初始化，因为与上电复位或系统复位相比，不需要 Flash 起动（~1 ms）和时钟 VCO/PLL 起动（~1-4 ms）的时间。
4. 通常，在复位测试阶段，需要检查所有 CPU 或处理器内部 I/O 模块定义的硬件中断或异常，或 CPU 或是处理器内部 I/O 模块定义的所有硬件的中断或是异常。发生上面情况时，会返回异常或非法中断的复位原因，并把异常报告给上层软件，用作诊断。
5. 有时外部电源产生的 PROST 复位，可能有进一步的原因，如电源欠压或外部看门狗溢。这需要从外部电源稳压器或是其它系统器件获取，相应的复位类型是硬件复位。PROST 的复位原因由更高层的应用程序分析处理。
6. 初步的复位测试在唯一 CPU（CPU0）上运行，测试过程中，其它 CPU 保持在复位状态。如果器件内有多个 CPU，不管是由哪个 CPU 调用，以上所有的接口都应报告相同的值。复位测试方法不具有故障检测功能，除非 Flash ECC 已经激活。
7. PROST 冷复位的情况下，应测试和初始化所配置的内存区。

3.3 初始化驱动和外设为默认设置

本节阐述了在启动阶段复位发生后，如何最快速激活外设的默认设置（与复位状态有所区别），并列举了一些典型示例。

1. 根据实际应用要求，把全部端口输入和输出信号设为所需的初始电气状态。基于基本的电气配置，初始化所有端口。例如：为启动加载程序和应用起动驱动程序,激活不同的片选信号。
2. 可能的话可激活另一个 CPU，而不再是 CPU0，来执行进一步的初始化动作。例如，一般只有 CPU1 带锁步，而默认 CPU0 不带，在功能安全相关的应用中，在初期安全测试执行前，最好先切换到带锁步的 CPU。
3. 这里，需要配置外部时钟输出的压摆率和驱动强度，以符合 EMI/EMC 要求。
4. 这部分将执行 ED 器件默认初始化，其初始化按照以下步骤进行：

- 检查单片机是否含 ED 器件，如果是，则开始时就对 ED 初始化。这里，通过寄存器 EMEM.CLC 使能 ED，激活时钟。通过寄存器 SBRCTR 启动解锁序列。这几步可以和 Cstart 阶段代码合并到一起。
- 仿真存储器应分配用于标定和跟踪功能。把没有用到的瓦片存储单元留作未使用的，为降低功耗，可利用门控将时钟关闭。在进行标定时，从上到下（T15, T14,然后是 T13，依此类推），初始化 TILECONFIG 寄存器。为了进行跟踪，从下到上（T0，T1，然后是 T2，依此类推），初始化瓦片存储单元。同时正确地配置 TILECC 和 TILECT 寄存器。TILESTATE 寄存器配置后，检测各自的瓦片式存储单元是否按正确的形式/模式进行分配。
- 对每个 CPU 的 32 重载区进行配置，把 Flash 区间（参考页）重载到仿真存储区间（工作页）。根据实际应用的需要，配置所有 96 个区间的目标地址，大小和重新定向的地址。重载的设置需要配置寄存器 RABRx, OMASKx 和 OTARTx。
- 重载区域配置完成后，就可以按照标定工具的请求，在这两个区域间切换（在参考页和工作页间），切换的操作通过寄存器 OVCCON 和 OVCENABLE 来控制。需要注意的是，需要在同一指令内，使数据缓存无效。通过 OSEL 配置也选择性地切换区间，在本文实现的示例中，没有使用这种方法。

3.4 启动阶段的安全测试和初始化

启动阶段，关键部件和基本 SMU 报警都要测试和初始化，如*启动安全测试*示图所示。START 驱动程序提供了钩子（Hooks），调用通用安全程序库或是应用相关的安全初始化函数，详细内容可参考 SafeTlib 文档。

1. 应用程序调用的前期安全测试钩子可以提供初始安全测试配置相关的信息，进而确定哪部分存储器和关键元件需要进行测试。
2. 预初始化确保先期运行的关键测试所需要的资源，变量是可用的，并且所需要的内存也经过了初始化。每个核共享的资源由主核初始化，而各核独自使用的资源，由相应的内核初始化。
3. 根据安全级别的要求，每个工作循环内都要对关键部件进行测试。先期运行的关键需要优化测试序列，以确保如*启动时的 SBST & MBIST* 图所要求的最短测试时间。
   a. CPU+锁步 – SBST，锁步比较器检查，潜在故障测试
   b. 关键静态内存 – 可配置的 MBIST 测试，ECC 故障，寻址故障
   c. 关键 FLASH – ECC 故障
   d. 存储保护单元（MPU）
4. 需要提供一种方法对处理器内核完整性进行测试，而且能满足微控器内的每个处理器和锁步核的测试可以分开独立执行。
5. 本文实现的示例中，START 驱动程序只对关键存储器的进行启动测试。对于内存的测试，可用 March，Checkerboard 或非凡转测试等算法，可最多对 16 个内存区进行测试。应用通过钩子程序，可以动态地启动或停用主要内存测试。一旦有错误发生，将抛出误异，返回错误发生的地址。内存 ECC 电路测试函数会在每个工作循环内，对内存存储纠错代码（ECC）检测电路测试一次。测试方法是对预存有 ECC 错误的内存区进行读操作，测试时会向 SMU 的触发 ECC 报警，但不会产生复位或是中断。缓存存储器区这时还不能启用，因为在内存测试过程中，缓存存储会被测试覆盖。
6. 每个工作循环内，Flash ECC 电路测试函数都要对 Flash 存储器纠错代码（ECC）检测电路测试一次。测试方法是使用预存有 ECC 错误的 Flash 区，测试过程中，SMU 的 ECC 报警不会产生复位或是中断。
7. 驱动初始化和多核启动后，会执行功能安全初始化，包括 SMU 初始化，SMU 激活和安全看门狗初始化。进一步说，包括初始化 SMU，设置错误引脚和把 SMU 切换到运行状态。其实，功能安全测试和初始化的顺序，是在驱动初始化/多核启动之前还是之后，需要从系统层面，综合考虑。
8. 最后，通过多次调用服务函数，指定不同参数，执行不同的安全预运行测试，可以完成对不同功能模块的测试，特定报警测试也会执行。一些预运行测试，可在 OS 运行之前或之后执行，典型例子是对 OS 用到的资源的测试，如 CPU 的存储保护单元（MPU）,总线的 MPU，中断路由。所有预运行测试会生成签名，可用来判断这些单元的逻辑流是否正确。上层程序提供一个输入种子，以生成测试签名，这样能保证测试签名是动态值，而不是固定旧数据（避免粘滞故障）。另外，所有预运行测试产生的测试结果，可被测试通过/失败标准用作失效判断。
9. 要求带存储保护的测试只能在 OS 启动后运行。通常假设，在 OS 启动前，测试执行时，中断全部关闭。安全测试完成后，基本的存储访问保护机制（基于主 ID）才能初始化，避免由非安全的软硬件组件使用导致系统崩溃。

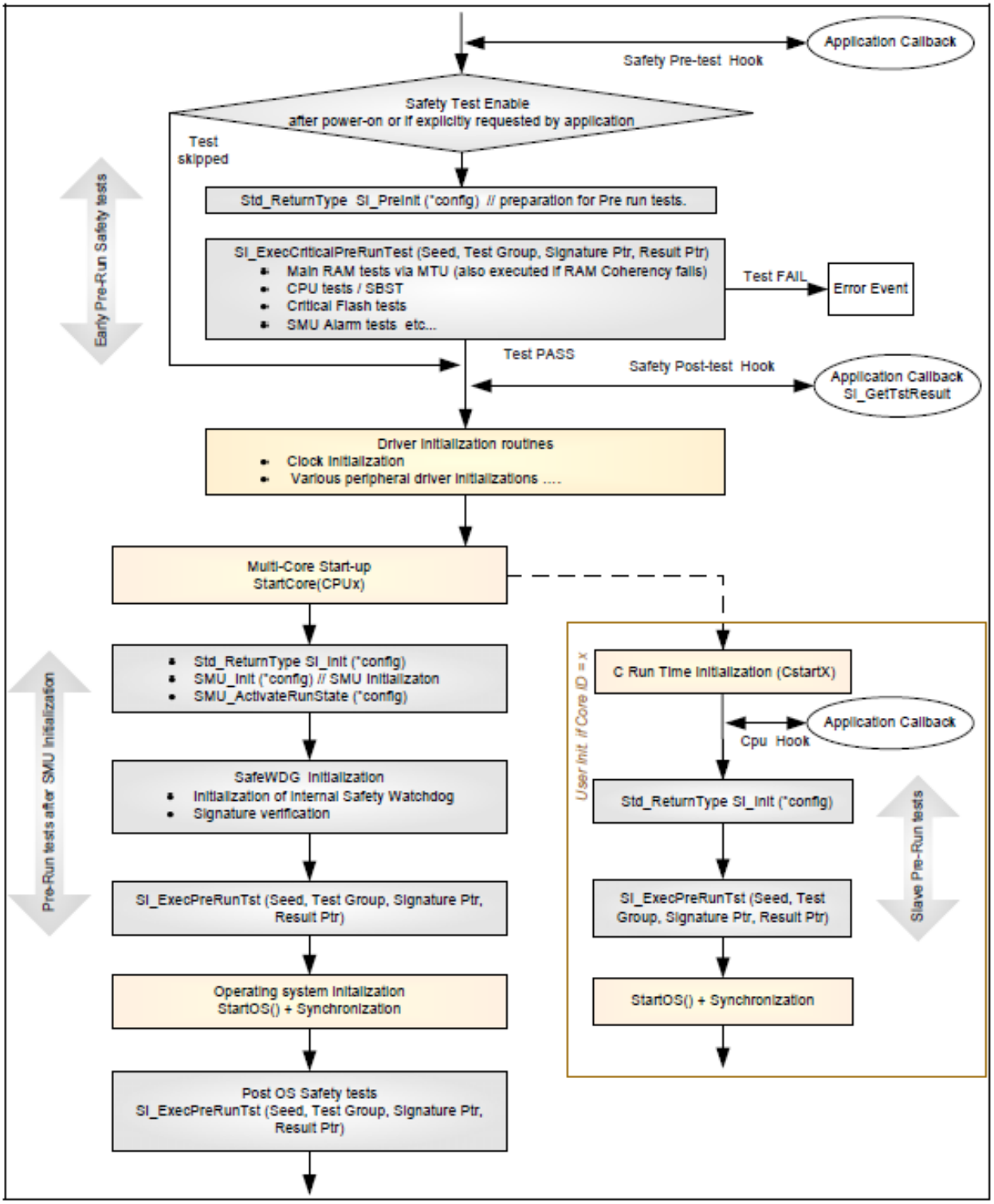

Application Callback
Safety Pre-test Hook
Safety Test Enable
after power-on or if explicitly requested by application
Test
skipped
Early Pre-Run Safety tests
Std_ReturnType SI_PreInit (*config) // preparation for Pre run tests.
SI_ExecCriticalPreRunTest (Seed, Test Group, Signature Ptr, Result Ptr)
Main RAM tests via MTU (also executed if RAM Coherency fails)
CPU tests / SBST
Critical Flash tests
SMU Alarm tests etc...
Test FAIL
Error Event
Test PASS
Safety Post-test Hook
Application Callback
SI_GetTstResult
Driver initialization routines
Clock initialization
Various peripheral driver initializations ....
Multi-Core Start-up
StartCore(CPUx)
Pre-Run tests after SMU Initialization
Std_ReturnType SI_Init (*config)
SMU_Init (*config) // SMU Initializaton
SMU_ActivateRunState (*config)
SafeWDG Initialization
Initialization of Internal Safety Watchdog
Signature verification
SI_ExecPreRunTst (Seed, Test Group, Signature Ptr, Result Ptr)
Operating system initalization
StartOS() + Synchronization
Post OS Safety tests
SI_ExecPreRunTst (Seed, Test Group, Signature Ptr, Result Ptr)
User Init. if Core ID = x
C Run Time Initialization (CstartX)
Application Callback
Cpu Hook
Std_ReturnType SI_Init (*config)
SI_ExecPreRunTst (Seed, Test Group, Signature Ptr, Result Ptr)
StartOS() + Synchronization
Slave Pre-Run tests

图 8 启动时的安全测试

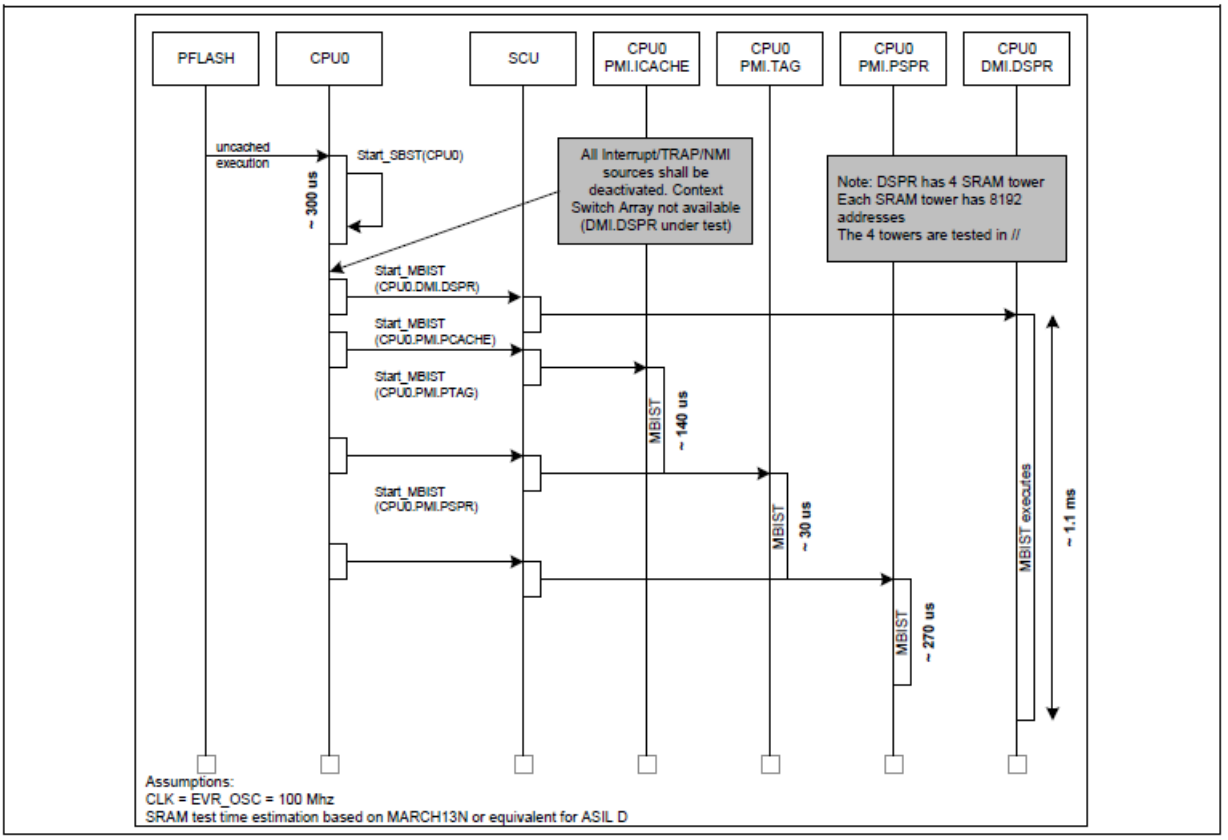


图 9 启动时的 SBST 和 MBIST

3.5 驱动程序初始化函数

3.5.1 时钟初始化

复位后，TC2xx 使用内部时钟开始运行，CPU 运行在 100 MHz，外设工作在 50 MHz。驱动程序初始化阶段，使用晶振或其它谐振器等作为外部时钟源，对锁相环（PLL）进行配置。

1. START 驱动程序需要实现的功能，包括使能，初始化，和把时钟分配给不同模块（如，CPU 时钟，总线时钟，预分频器，倍频器等，它们在微控器中可以进行配置）。时钟的配置过程包括初始化 PLL 比例因子，启动 PLL 锁进程，等待 PLL 锁定。配置时钟各个步骤，应避免大电流跳跃，且电流上升的级数是可配置的。PLL 配置完成后，不同模块分频系数和时钟（KSCFG 寄存器）基于 PLL 的配置进行初始化。

2. 在配置时钟和 PLL 前，需要检查当前时钟的状态，避免时钟配置好后，再重新配置。当前时钟实际配置要么是用户配置，复位配置或是未定义的，对于后两个配置，时钟应重新初始化。

3. 通过某种方法可以得到 PLL 状态，驱动程序需要提供查询微控器内全部 PLL 锁的状态的方法，通过查询得到锁定，未锁定或是不支持等返回信息，这点与 AUTOSAR API 是一样的。

4. 经过配置，频率调制被激活。N 分频器设成不同的值，对应可以得到不同非整形的频率，N 分频器使用不同的分频系数，就可以将频率调整到更快。使用这些方法，需要提供对应的配置参数，如使能，fm depth%，调制的目标频率。

5. SMU SafeTlib 具有时钟监控，晶体或振荡器故障检测，失锁以和时钟错误通知等时钟安全特性。还有，START 驱动程序需要支持时钟故障中恢复机制，故障发生时切换到备份时钟，如 100MHz 的内部时钟。

6. 需要注意的是，驱动程序应该给出适当的等待时间，如循环等待 VCO 锁住事件，Kx 分频系数更新等。在更新两个 K2 分频系数之间时，至少需要等待 6 个 fPLL 周期。PLL 锁定时间，数据手册上给出的范围是 14us 到 200us 间。

7. 当重要时钟的频率相对标称值的偏差超过 ± 5.5%时，可以认为是时钟故障。当失效发生时间间隔超过 100ms，故障处理函数将被调用，进行时钟恢复。时钟监控功能要确保发生时钟共因失效时，所有功率输出接口要在故障出现 200 ms 内恢复成复位状态输出。SMU SafeTlib 软件可以对失效进行处理。为与 AUTOSAR 接口兼容，AUTOSAR DEM 驱动程序的应该包含对时钟失效的处理。如果没有使用，可对 START_SetEX_Error 事件（带 CLOCK_FAILURE 参数）进行处理。对于失锁事件，也会报告 DEM 失效。

8. 基于用户配置，配置 flash 访问的等待状态，因为这取决于时钟设置。激活预取指缓冲和跳转预测器。完全激活高速缓存，以达到最高的性能。

9. 如果应用需要，可打开外部时钟输出（EXTCLK0 & EXTCLK1），该功能是可静态配置。以下参数也应是可配的：时钟分频器，时钟源的使用： fspb→fout

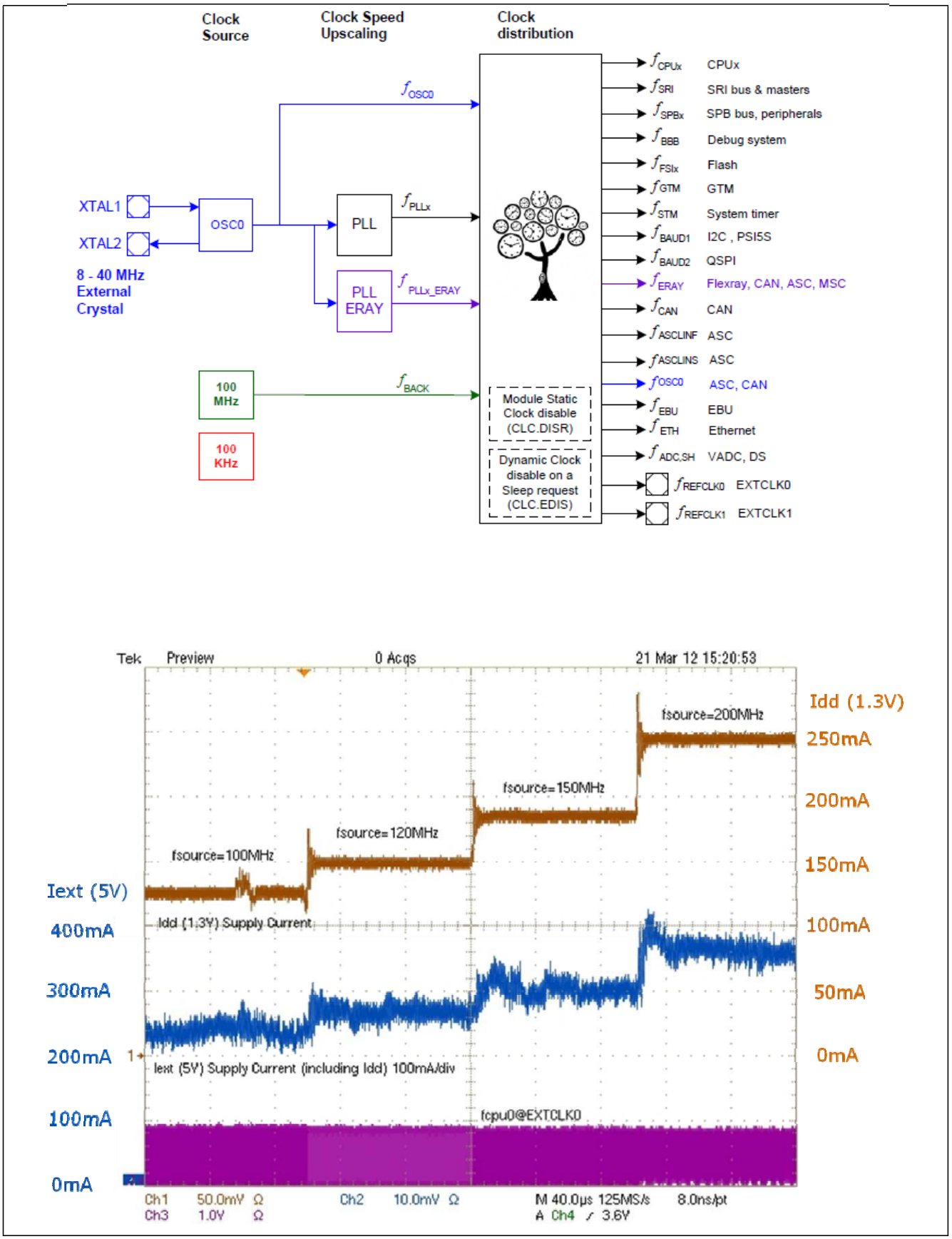


图 10 时钟起动 （50 mA/100 us）

表 2 典型时钟分频器推荐

CCU Clock Dividers	Min / Max	TC29x 300 MHz	TC27x/6x 200 MHz	TC26x 160 MHz	TC24x 133 MHz	TC24x 80 MHz
MAXDIV	TC29x:300MHz TC27x:200MHz TC26x:200MHz TC24x:133MHz	=SRIDIV	=SRIDIV	=SRIDIV		=SRIDIV
OSC0 crystal	16 or 20 MHz 8 MHz-48 MHz	20MHz	20MHz	20MHz		20MHz
VCO (N,P,K2)	VCO: 400-800 MHz	fvco=600 MHz P=2 N=60 K2=2 K2_1=6 K2_2=4 K2_3=3	fvco=600 MHz P=2 N=60 K2=3 K2_1=6 K2_2=5 K2_3=4	fvco =640 MHz P=2 N=64 K2=4 K2_1=6 K2_2=5 K2_3=4		fvco =400 MHz P=2 N=40 K2=5 K2_1=4 K2_2=5 K2_3=5
SRIDIV	Same as MAXDIV	300 MHz	200 MHz	160 MHz		-
SPBDIV	Max : 100 MHz	50 MHz / 100 MHz	40 MHz / 100 MHz	40 MHz/ 80 MHz fethram = 160 MHz		40 MHz / 80 MHz fethram = 160MHz
CPU0DIV	Same as MAXDIV	200 MHz	200 MHz	150 MHz/ 160 MHz		80 MHz
CPU1DIV	Same as MAXDIV	300 MHz	200 MHz	75MHz/80MHz		
CPU2DIV	Same as MAXDIV	300 MHz	200 MHz	-		
VCO Eray (N,P,K2)	VCO: 400 - 480 MHz 16-24 MHz OSC0	400 MHz	400 MHz	400 MHz		400 MHz
ERAYDIV	Max: 80 MHz	80 MHz	80 MHz	80 MHz		80 MHz
GTMDIV	Max: 100 MHz	50 MHz / 100 MHz	40 MHz / 100 MHz	40 MHz / 80 MHz		40 MHz / 80 MHz
STMDIV	Max: 100 MHz	50 MHz / 100 MHz	40 MHz / 100 MHz	40 MHz / 80 MHz		40 MHz / 80 MHz
BAUD1DIV	Max: 100 MHz	50 MHz / 100 MHz	40 MHz / 100 MHz	40 MHz / 80 MHz		40 MHz / 80 MHz
BAUD2DIV	Same as MAXDIV	50 MHz / 100 MHz	40 MHz / 100 MHz	40 MHz		40 MHz
CANDIV	Max:100 MHz	50 MHz / 100 MHz 80 MHz via feray	40 MHz / 100 MHz 80 MHz via feray	40 MHz 80 MHz via feray		40 MHz
ASCLIN FDIV	Same as MAXDIV					
ASCLIN SDIV	Max:100 MHz	20 MHz	20 MHz	20 MHz		20 MHz
ETHDIV	Max:50 MHz		25MHz MII / 50MHz RMII	50 MHz RMII		50 MHz RMII

CCU Clock Dividers	Min / Max	TC29x 300 MHz	TC27x/6x 200 MHz	TC26x 160 MHz	TC24x 133 MHz	TC24x 80 MHz
EBUDIV	Same as MAXDIV	200 MHz	-	-		-
K3 + ADCCLKSEL	9x&7x:133MHz 6x&4x:80 MHz	133 MHz (ErayPLL)	133 MHz (ErayPLL)	80MHz (ErayPLL)		80MHz
VADC (DIVD)	limited to fSPB	fspb	fspb	fspb		fspb
VADC (DIVA)	limited to fSPB	20 MHz (fspb/x)	20 MHz (fspb/x)	20 MHz (fspb/x)		20 MHz (fspb/x)
DS ADC	limited to fSPB	20 MHz (feray/4)	20 MHz (feray/4)	20 MHz (feray/4)		20 MHz (feray/4)
BBBDIV	fspb upto fsri/2	fsri/2	fsri/2	fsri/2		fsri/2
FSIDIV	Max:100 MHz	fsri/3	fsri/2	fsri/2		fsri
FSI2DIV	Same as MAXDIV	fsri	fsri	fsri		fsri
MSC ABRA	Same as MAXDIV	40 MHz / 80 MHz (feray)	40 MHz / 80 MHz (feray)	40 MHz / 80 MHz (feray)		40 MHz / 80 MHz
HSM	fspb	fspb	fspb	fspb		fspb
WSPFLASH, WSECPF, WSDFLASH, WSECDF		t.b.d	t.b.d	t.b.d		t.b.d

3.6 多核启动

各驱动程序初始化完成后，START 驱动触发系统初始化事件，应用程序有机会通过钩子 StartSysInitHook(START_EX_SysInit)运行初始化软件。

1. 到现在，硬件只启动了 CPU0 核，而其它核仍保持在暂停（Halt）状态，软件通过寄存器 DBGSR，可以激活其余内核。可以配置 CPU 激活以及相应的起始地址。
2. 每个核都都可以读到自己所处的状态，是四种状态中的一个 - 暂停，空闲或是运行。通过函数 getCoreMode 和 setCoreMode，可得到当前状态，切换状态。通过 getCoreId 方法，可以得到内核的 ID。
3. 在把 CPU 状态从暂停置为运行前，须设置好 CPU 的程序计数（PC），可通过 setCorePC 读到。
4. 如果激活了一个核，在“主”函数调用前，核会执行最主要单独的 C 运行时间环境初始化。每个核执行的是相同“主”函数，则须在函数中通过各自核 ID 来区分各个核。
5. 之后，应用通过钩子 StartCPUxInitHook，执行各 CPU 具体的配置。
6. 如没之前 CPU1 或是 CPU2 未测试或被激活，我们需要将该 CPU 状态从暂停切换到运行，来进行激活。如果之前已激活，将 CPU 置为空闲模式，之后再切换到运行模式，重新激活。通过寄存器 DBGSR 可退出暂停状态，而进入/退出空闲状态，是通过寄存器 PMCSRx 来处理的。
7. 各 CPU 可以通过软件产生中断，并向其他 CPU 请求中断服务，且每个 CPU 都可以对 IO 中断源和 DMA 进行配置。
8. 核的激活也可以 OS 实现的方法来操作。

3.7 操作系统切换

使用钩子 StartSysInitHook (START_EX_SysInit)，可实现 OS 相关的配置。此时，所有 OS 用到的基本软件模块都要被初始化。在完成 OS 初始化之前，如果需要使用中断，只有 I 类中断能在 STARTUP 驱动阶段使用。可以通过预编译的配置选项，确定是否配置和编译这部分程序。

1. 系统服务 StartOS 执行以下动作：
   - 执行操作系统初始化代码
   - 执行 StartupHook（ISR II 类中断禁止）。
   - 使能所有中断，启动调度，启动自动运行的任务。
   - 初始化信号量和自旋锁等机制的默认状态。
2. 在 OS 内处理，进一步处理定时和存储保护功能，启动代码部分没有初始化内部存储的保护功能，如存储保护和访问保护。信号量和自旋锁机制可用于多核 OS 系统的临界区，存储区或资源的锁定和解锁。
3. 多核配置中，在主核调用 StartOS 之前应激活每个处于停止状态的核。而且，StartOS 要被所有内核调用。在内核首次同步之后，OS 执行应用程序的启动钩子程序，之后内核可能再次同步。

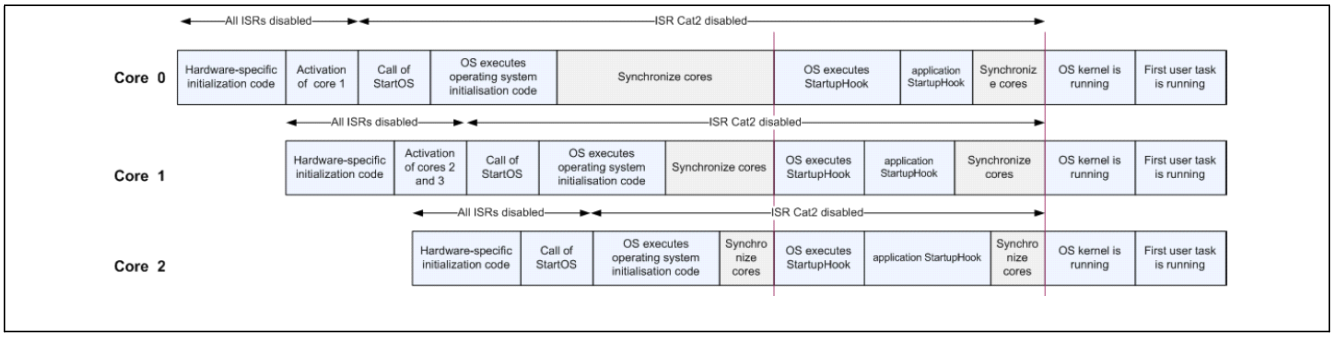


图 11 OS 初始化和同步

4 TC27xC 启动

4.1 CPU 中断处理

TriCore 架构的中断管理给中断延时的最小化提供了硬件支持，如由硬件自动保存/恢复中断上下文，中断确认后清除服务请求标志（SRR），以及进入 ISR 入口时关全局中断等。中断路由模块中与服务提供者都对应有一个中断控制单元（ICU），它对服务请求节点发出的服务请求进行仲裁，并且把仲裁结果提供给服务提供者，如图*中断处理*所示。每个服务提供者（CPU 或 DMA）可响应所有 512/1024 中断服务请求中的最多 255 个中断或优先级。

1. 每个 CPU 中都有自己的中断向量表，而每个中断服务函数都有对应的中断请求优先级数值，CPU 中断向量表本身就是按优先级进行排列。当响应一个中断时，CPU 用中断优先级数值作为偏移量，与中断向量的基地址，可以得到唯一的指向该向量位置的 PC 地址，如图*中断处理*中 jump①所示。因此， CPU 中断向量表允许一个外设有多个优先级以满足不同用途。

2. 中断向量表可以在链接时静态安装，或通过软件中断安装处理函数动态安装，把中断服务函数（ISR）与向量表中断入口（SRPN 优先级）相关联。从而确保向量表中断入口指向响应服务函数（ISR）的起始位置，如图*中断处理*中，可以产生跳转 jump②。实际系统可以有多个中断向量表，通过改变 BIV 寄存器可以实现在不同向量表间的切换，就像在 OS 所 做的那样。另外，如果 255 个中断已经能够满足系统的需求，所有 CPU 也可以共用一个向量表。在这种情况下， CPU0_BIV, CPU1_BIV 和 CPU2_BIV 将初始化指向同一向量表 INTTAB 的位置。

3. 复位后，中断向量表的基地址置默认置为 BIV=0000 0000h，在系统启动阶段通过指令 MTCR 进行初始化设置。向量表的第一个入口（SRPN=0000h）被预留且不能用作 CPU 响应的服务请求。BIV 基址必须与偶数字节/半字地址边界对齐。

4. 复位后，所有服务请求节点默认为禁用。如果要使用某个服务请求节点，必须通过把相应节点的 SRC.SRE 位置为 1 来配置和使能。如果 SRE = 1，等待的服务请求会加入到的服务提供者 ICU 的中断仲裁中，具体是哪个服务提供者响应中断通过 TOS 位字段选择。在修改 TOS 或 SRPN 位字段之前，相应的服务请求节点（SRN）必须设为禁用（SRE=0），可以分两步/两次写操作清除挂起的服务请求标志 SRR，使能相应的服务请求节点（SRN），如下：首先清除 SRR 标志（SRC.CLRR 置 1），然后使能（SRC.SRE 置 1）。

5. 通常，复位后全局中断处于关闭状态。在中断能被响应之前，需要使能全局中断。通过中断控制寄存器（ICR）中的全局中断使能位（ICR.IE），可使能 ICU 将中断请求传递到相应的 CPU。

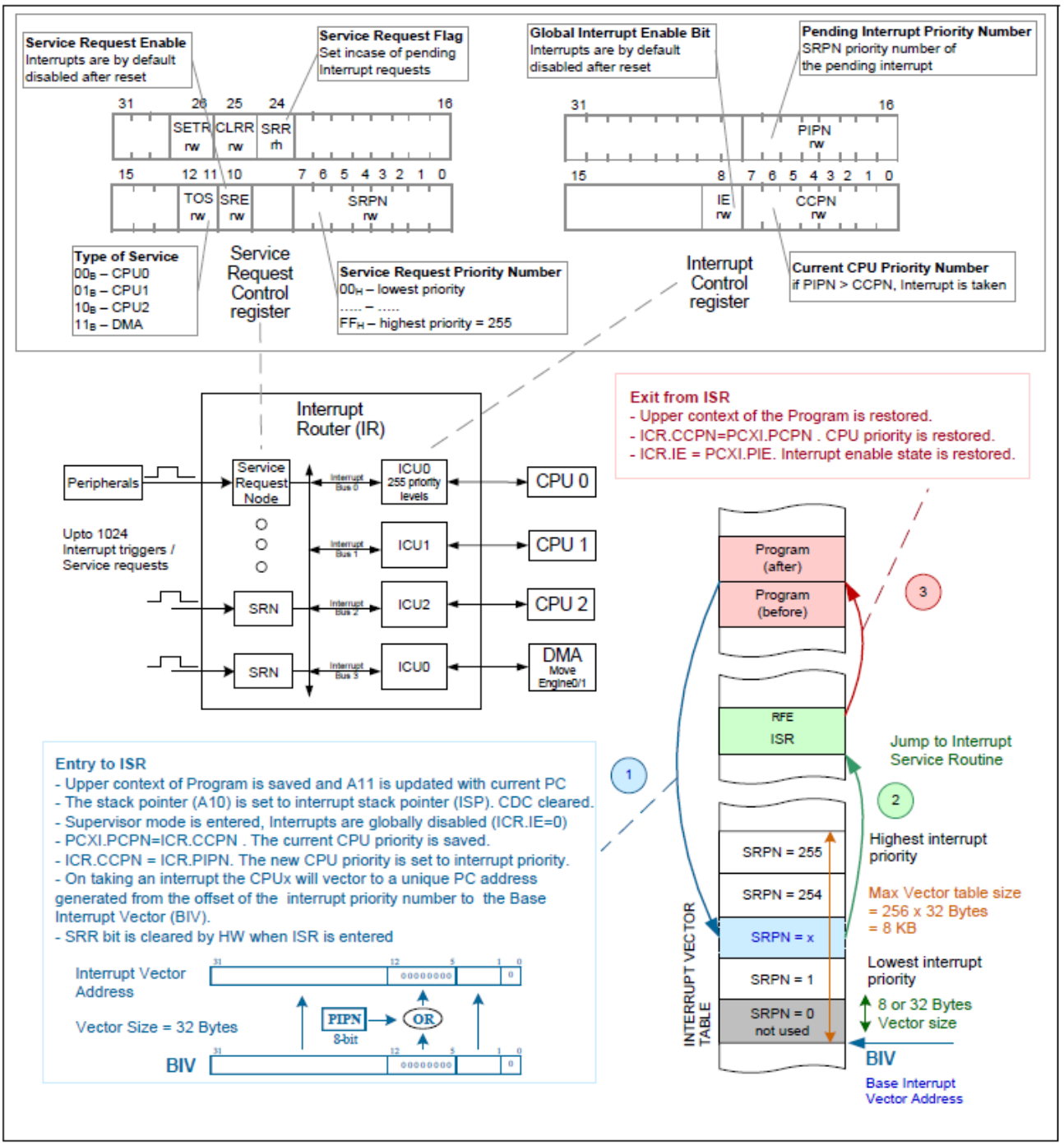

Service Request Enable
Interrupts are by default disabled after reset
Service Request Flag
Set incase of pending Interrupt requests
Global Interrupt Enable Bit
Interrupts are by default disabled after reset
Pending Interrupt Priority Number
SRPN priority number of the pending interrupt
SETR rw
CLRR rw
SRR rh
TOS rw
SRE rw
SRPN rw
PIPN rw
IE rw
CCPN rw
Type of Service
00B – CPU0
01B – CPU1
10B – CPU2
11B – DMA
Service Request Control register
Service Request Priority Number
00H – lowest priority
..... – .....
FFH – highest priority = 255
Interrupt Control register
Current CPU Priority Number
if PIPN > CCPN, Interrupt is taken
Interrupt Router (IR)
Peripherals
Service Request Node
ICU0 255 priority levels
ICU1
ICU2
ICU0
SRN
SRN
CPU 0
CPU 1
CPU 2
DMA Move Engine0/1
Upto 1024 Interrupt triggers / Service requests
Exit from ISR
- Upper context of the Program is restored.
- ICR.CCPN=PCXI.PCPN . CPU priority is restored.
- ICR.IE = PCXI.PIE. Interrupt enable state is restored.
Program (after)
Program (before)
RFE
ISR
Jump to Interrupt Service Routine
Highest interrupt priority
SRPN = 255
SRPN = 254
SRPN = x
SRPN = 1
SRPN = 0 not used
Max Vector table size = 256 x 32 Bytes = 8 KB
Lowest interrupt priority
8 or 32 Bytes Vector size
INTERRUPT VECTOR TABLE
BIV
Base Interrupt Vector Address
Entry to ISR
- Upper context of Program is saved and A11 is updated with current PC
- The stack pointer (A10) is set to interrupt stack pointer (ISP). CDC cleared.
- Supervisor mode is entered, Interrupts are globally disabled (ICR.IE=0)
- PCXI.PCPN=ICR.CCPN . The current CPU priority is saved.
- ICR.CCPN = ICR.PIPN. The new CPU priority is set to interrupt priority.
- On taking an interrupt the CPUx will vector to a unique PC address generated from the offset of the interrupt priority number to the Base Interrupt Vector (BIV).
- SRR bit is cleared by HW when ISR is entered
Interrupt Vector Address
Vector Size = 32 Bytes
PIPN
8-bit
OR
BIV
1
2
3

图 12 中断处理

表 3 中断初始化序列

1.)	初始化中断向量基址
	
	a）32 字节向量间隔 - BIV 基址最好与 32 字节边界对齐， BIV.VSS 位清零。
	例如：- ___INTTAB_CPU0 =0x800F0000; // defined in lsl file
	___mtcr (BIV, ___INTTAB_CPU0); // initialized in cstart file
	
	b）8 字节向量间隔 - BIV 基址最好与 8 字节边界对齐，BIV.VSS 置 1。
	例如：- ___INTTAB_CPU0 =0x800F0001; // defined in lsl file
	_mtcr (BIV, ___INTTAB_CPU0); // initialized in cstart file
	
	c）单入口 - 将寄存器 BIV 配置为普通单入口方式，入口位置放置中断函数处理，通过一组函数指针就可跳转到相应的中断服务程序。需要屏蔽 PIPN 字段，这样任一中断计算结果都是相同地址。
	例如：- ___mtcr(BIV,0x80000000 | 0xFF<<5); // BIV=0x80001FE0 // incase of BIV.VSS=0
	例如：- ___mtcr(BIV,0x80000001 | 0xFF<<3); // BIV=0x800007F9 // incase of BIV.VSS=1

2.a)	链接时向量表中断处理函数的安装 这取决于所选择的链接器/编译器，并在相应链接脚本文件内定义。中断向量表中的中断入口在连接时被初始化为中断服务程序的入口地址。

2.b)	软件管理安装向量表中断处理函数
	a)void (*isr_pointer_array[256])(void);//define a ISR pointer array or interrupt vector table.
	b)void interruptHandlerInstall (uint32 srpn,unint32 addr){ *isr_pointer_array[srpn]=addr; }
	运行时，通过以上函数将中断向量表中断入口初始化为 ISR 入口地址。

3.	定义中断服务程序
	a) ___interrupt can0_isr() { _enable(); // CAT1. frame….
	/*some user application code */
	asm (" rfe"); } // Macros from compiler maybe used.
	b) ISR can0_isr() { OS frame entry // CAT2.OS preamble
	/*some user application code */ call to application routine
	OS frame exit // incl. OS task dispatch
	asm (" rfe"); // ISR macro may encapsulate additional OS specific frame or preamble/ postamble before entering the application routine.

4	配置服务请求节点 a) src_init (uint32 *src, uint32 tos, uint32 srpn); // 服务请求与中断控制单元或服务它提供者(CPU0 或 CPU1 或 CPU2 或 DMA)关联，服务请求赋予固定的优先级，更新寄存器 SRC 的 TOS 和 SRPN 字段。
5	激活服务请求节点，使其中请求能被对应 ICU 仲裁和响应 a) src_enable (uint32 *src); // The SRC.SRE 位置位。
6	打开全局中断 a) _enable; // The ICR.IE 位置位。

4.2 CPU 陷阱处理

陷阱（trap）发生是由不可屏蔽中断（NMI）、异常指令、_syscall ()或非法访问等事件导致的。所有陷阱被分成 8 陷阱类（TCN），按属性也可分为同步、异步、硬件和软件陷阱。

1. 按陷阱类别编号的形式，将陷阱服务程序（Trap Service Routines）与各个 CPU 特定的陷阱向量表关联起来，陷阱向量表按陷阱类别编号进行排序，编号可用作陷阱向量表索引。每个陷阱类别分配一个陷阱服务程序（TSR），一旦发生该类别陷阱，便会执行相应程序来提供相关服务。如陷阱处理图中 jump①所示，当陷阱产生时，CPU 以陷阱类别编号（TCN）作为偏移，与陷阱向量基址（BTV）计算出唯一 PC 指向向量表中入口的位置。陷阱不会改变触发该陷阱的程序的 CPU 中断优先级，所以 ICR.CCPN 字段不会改变。当陷阱发生时，CPU 会中断正在执行的指令，并且强制执行相应的 TSR。另外，每个陷阱还会分配一个陷阱识别码（TIN），通过 TIN 可以查询发生该类陷阱的原因。在陷阱服务程序（TSR）入口，陷阱识别码（TIN）便会加载到数据寄存器 D[15]内。在 TSR 时，通过 D[15]以得到造成陷阱的原因。
2. 陷阱服务程序可作为普通函数来实现， 通过 OS 或是编译器相应配置文件的定义，安装在陷阱向量表内。如果实际应用中使用 OS 时，要注意的是，一般将陷阱处理放在 OS 的实现里。下面列举了各 CPU(x= 0,1,2)的相应 TCN 类别的陷阱函数。
   1. CoreX_tsrMmu()
   2. CoreX_tsrProtection ()
   3. CoreX_tsrInstruction()
   4. CoreX_tsrContext()
   5. CoreX_tsrBus ()
   6. CoreX_tsrAssertion ()
   7. CoreX_tsrOSSystemCall()
   8. CoreX_tsrNMI()
3. 陷阱向量表可在链接时静态安装，或用陷阱安装处理函数通过软件动态安装，从而把 TSR 函数和向量表入口/陷阱类别相对应，确保向量表的中断入口指向相应 TSR 的起始地址，如*陷阱处理*图中 jump②。各 CPU 可能有各自的陷阱向量表，所有 CPU 也可以共用同一个的陷阱向量表，然后在 TSR 内，根据具体的内核再跳转。这时，CPU0_BTV, CPU1_BTV 和 CPU2_BTV 初始化后，将指向同一个向量表___TRAPTAB 的位置
4. 复位之后陷阱向量表的基址默认为 BTV = A000 0100h，在启动阶段初通过指令 MTCR 初始化。向量表位于非高速缓存区，可以对高速缓存陷阱作出响应。由于不支持 MMU，向量表第一个入口（TCN = 00h）被预留。BIV 基址必须与 256 字节地址边界对齐。
5. 陷阱始终为有效，不能被停用。同步陷阱发生在执行某种特定指令时，并在执行下一条指令之前先服务陷阱。异步陷阱和中断类似，在内核外部检测到陷阱，并把陷阱信号传递给内核。软件陷阱可用作系统调用，断言陷阱或是 NMI。

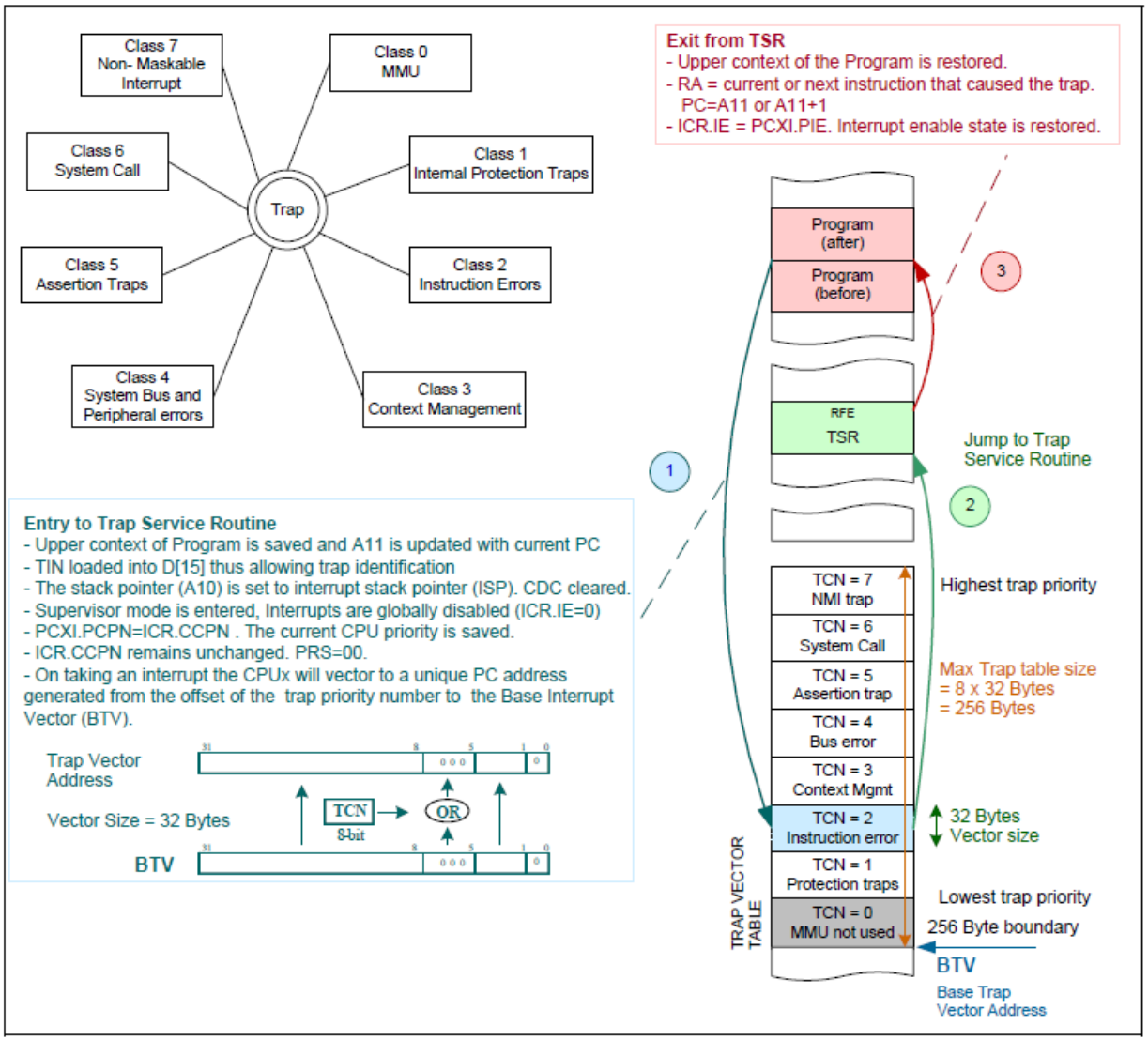


图 13 陷阱处理

表 4 陷阱初始化一般序列

1.)	陷阱向量基址初始化
	
	a）32 字节向量间隔 – BTV 基址与 256 字节边界对齐。
	Eg.– ___TRAPTAB_CPUx =0xA000 0100; // defined in lsl file
	____mtcr (BTV, ___TRAPTAB_CPUx); // initialized in cstart file

2.a)	链接时安装向量表中断处理函数 这取决于所选的链接器/编译器，并在相应的链接脚本内定义。链接时，陷阱向量表入口被初始化为 TSR 入口地址。

2.b)	安装软件管理向量表陷阱处理函数
	a) void (*tsr_pointer_array[8]) (void); //define a TSR pointer array or trap vector table.
	b) void trapHandlerInstall (uint32 tcn, unint32 addr) { *isr_pointer_array[tcn]=addr; }
	通过以上函数会安装陷阱处理函数，也就是陷阱向量表中，对应类型的陷阱入口指向陷阱服务程序(TSR)的入口。

3.	陷阱服务程序定义
	a) int ___trap(7) Cpu_trapClass7(void) /* Trap class 7 handler.*/
	{ Cpu_tsrNonMaskableInterrupt();
	asm (" rfe"); } // Macros from compiler maybe used.
	b) TSR (7) { IfxCpu_tsrNonMaskableInterrupt();
	asm (" rfe"); } // OS TSRs maybe used.
	开发阶段，默认陷阱程序包括_debug()函数，用来简化陷阱调试。

4.3 次级引导程序处理

通常，次级引导加载程序和应用分别有各自的启动驱动程序。

1. 启动阶段需要评估，该执行次级引导程序，还是应用程序驱动，通过“application presence”标志和相应签名确定要执行哪段程序。如果存在应用程序，则直接跳转到应用实例，相反则跳转到引导加载程序。而标志变量一般存储在 NVM（存储器-Dflash 或外部 EEPROM），引导加载程序和应用程序都可以改写标志。
2. 如果引导加载程序执行了内存测试，通常应用程序内就不再做该测试，反之亦然。
3. 如果次级引导加载使用了 CAN，则在引导加载程序内需要配置时钟。引导加载程序还可能用到的其它驱动包括 ：CAN, SCI, Flash, Digin, Digout, SPI, IRQ。
4. 通常，引导加载程序采用固定时间的任务调度。
5. START_JumpToAppl (..)函数可将程序从引导加载部分，跳转到应用程序部分。程序跳转前，应用程序应当注意：
   - 跳转前，引导加载程序应关掉中断和 DMA。
   - 引导加载程序用到的模块应置回到复位状态。

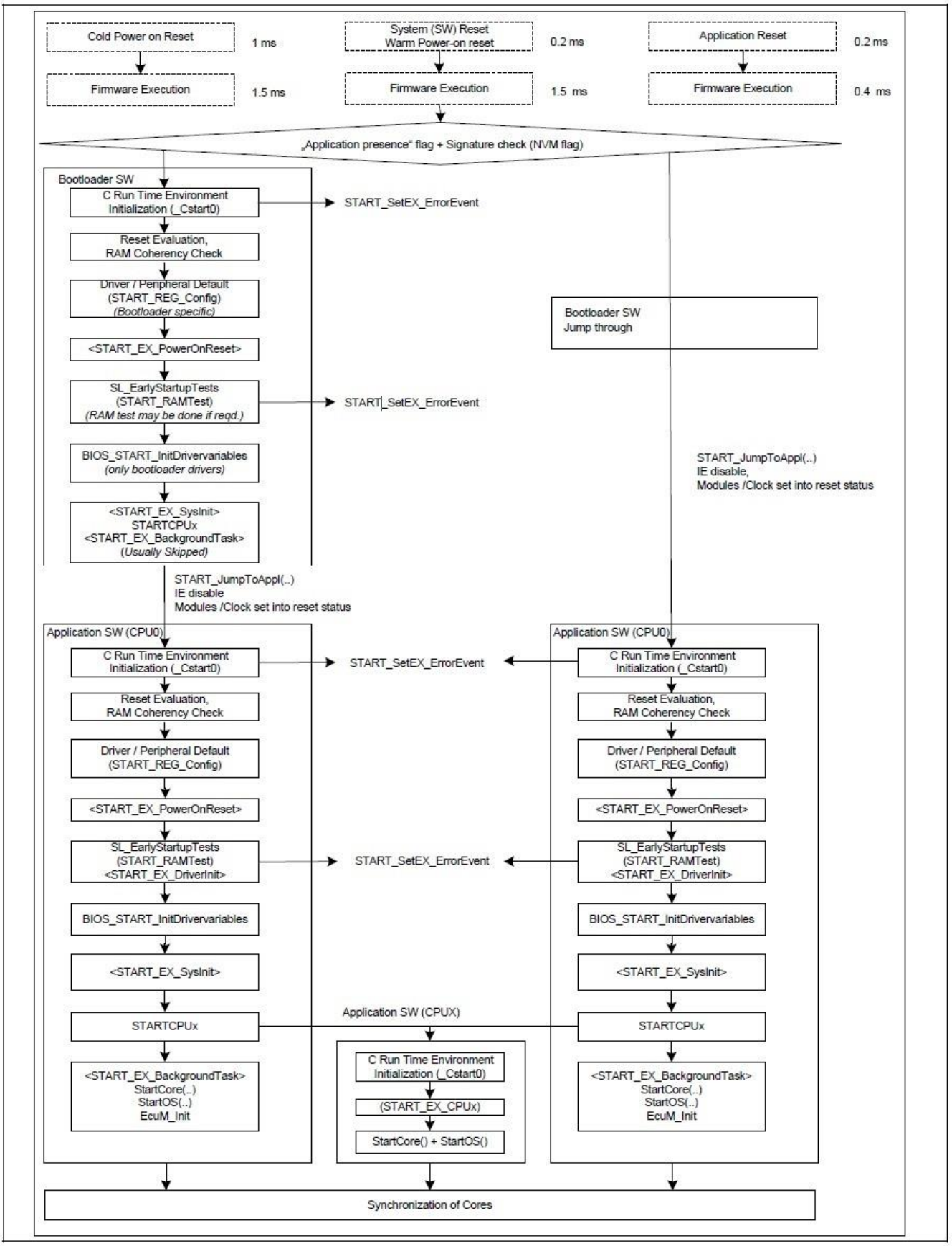

Cold Power on Reset
1 ms
System (SW) Reset
Warm Power-on reset
0.2 ms
Application Reset
0.2 ms
Firmware Execution
1.5 ms
Firmware Execution
1.5 ms
Firmware Execution
0.4 ms
„Application presence" flag + Signature check (NVM flag)
Bootloader SW
C Run Time Environment Initialization (_Cstart0)
START_SetEX_ErrorEvent
Reset Evaluation, RAM Coherency Check
Driver / Peripheral Default (START_REG_Config) (Bootloader specific)
<START_EX_PowerOnReset>
SL_EarlyStartupTests (START_RAMTest) (RAM test may be done if reqd.)
START_SetEX_ErrorEvent
BIOS_START_InitDrivervariables (only bootloader drivers)
<START_EX_SysInit> STARTCPUx <START_EX_BackgroundTask> (Usually Skipped)
Bootloader SW Jump through
START_JumpToAppl(..) IE disable, Modules /Clock set into reset status
START_JumpToAppl(..) IE disable Modules /Clock set into reset status
Application SW (CPU0)
C Run Time Environment Initialization (_Cstart0)
START_SetEX_ErrorEvent
Reset Evaluation, RAM Coherency Check
Driver / Peripheral Default (START_REG_Config)
<START_EX_PowerOnReset>
SL_EarlyStartupTests (START_RAMTest) <START_EX_DriverInit>
START_SetEX_ErrorEvent
BIOS_START_InitDrivervariables
<START_EX_SysInit>
STARTCPUx
<START_EX_BackgroundTask> StartCore(..) StartOS(..) EcuM_Init
Application SW (CPU0)
C Run Time Environment Initialization (_Cstart0)
Reset Evaluation, RAM Coherency Check
Driver / Peripheral Default (START_REG_Config)
<START_EX_PowerOnReset>
SL_EarlyStartupTests (START_RAMTest) <START_EX_DriverInit>
BIOS_START_InitDrivervariables
<START_EX_SysInit>
STARTCPUx
<START_EX_BackgroundTask> StartCore(..) StartOS(..) EcuM_Init
Application SW (CPUX)
C Run Time Environment Initialization (_Cstart0)
(START_EX_CPUx)
StartCore() + StartOS()
Synchronization of Cores

图 14 二级引导程序

www.infineon.com

Infineon Technologies 出版发行