1.1 关于内核程序

VxWorks内核应用程序以与内核本身相同的模式和内存空间执行。在这方面，它们不同于与其他操作系统（如UNIX和Linux）的应用程序，也不同于VxWorks实时（RTP）应用程序。

内核应用程序可以交互式下载并在VxWorks目标系统上运行，或者与操作系统映像链接并（可选地）配置为在引导时自动执行。

在内核中执行的VxWorks应用程序被创建为可重新定位的对象模块。它们可以更具体地称为基于内核的应用程序模块，但是像本指南中所做的那样，简单地将它们称为内核应用程序模块或内核应用程序通常比较方便。（它们有时在Workbench项目类型可下载内核模块的缩写之后称为DKM，即使它们与内核链接时也是如此。）内核应用程序不应该与作为实时进程（RTP）在用户模式下执行的应用程序混淆。

当构建基于内核的应用程序模块时，将用户代码链接到所需的VxWorks库，并生成ELF二进制文件。内核应用程序使用VxWorks工具，包括定义操作系统接口和数据结构的头文件。

内核应用模块可以是：

通过对象模块加载程序下载并动态链接到操作系统。

与操作系统静态链接，使它们成为系统映像的一部分。

内核模块对于快速开发和调试很有用，因为不需要为应用程序的每次迭代重新构建操作系统映像。该方法也可用于生产系统的诊断设备。各种开发工具，包括调试器和shell（主机或内核），可以用来下载和管理模块。模块可以从内核支持的任何主机文件系统（NFS、FTP等）下载到目标。

内核应用模块也可以存储在Flash或ROM、ROMFS文件系统或磁盘上。一旦加载到目标中，内核应用程序模块就可以从shell或Workbench交互式地启动。

静态链接到操作系统的应用程序模块可以与shell或Workbench交互运行。VxWorks还可以被配置为在启动时自动启动它们。静态连接和自动启动显然适用于生产系统。

在内核空间中运行的应用程序不作为进程执行；它只是在内核空间中运行的另一组任务。内核不保护内核应用程序的任何不当行为的保护，并且应用程序也同样不保护其他内核应用程序的保护，内核模块在超级用户模式下在相同的地址空间中运行。

警告：如果希望将内核应用程序移植到作为实时进程（RTP）执行的用户模式应用程序，则必须确保它满足RTP应用程序的代码要求，并按此进行编译。还必须确保VxWorks配置为对RTPs的支持。有关更多信息，请参见22，内核到RTP应用程序迁移。

内核应用和内核组件需求

VxWorks是一个高度可配置的操作系统。当构建独立于操作系统内核应用程序模块时，构建过程无法确定应用程序最终将在其上运行的VxWorks实例是否已经配置了应用程序所需的所有组件（例如，网络和文件系统）。因此，应用程序代码检查指示内核设施不可用的错误（即，检查API调用的返回值）并做出适当响应是很有用的。当内核应用程序模块与操作系统链接时，构建系统生成关于丢失组件的错误。Workbench和wrtool命令行工具都提供了检查依赖关系并相应地重新配置VxWorks的机制。

二进制文件兼容性

注意：为VxWorks构建的代码只有在基于相同的VSB配置（具有相同的层和版本集）时才是二进制兼容的。此外，内核C++代码必须用与VxWorks映像相同的编译器来构建。

注意：如果模块与VxWorks系统不兼容，内核对象模块加载器将拒绝该模块，打印错误消息，并设置errno（当体系结构不同时设置S\_loadElfLib\_HDR\_READ，或者当其他特性不同时设置S\_loadLib\_INCOMPATIBLE\_MODULE）。

内核应用程序与RTP应用程序的比较

内核应用程序和RTP应用程序之间的主要区别在于，RTP应用程序在单独的受保护的内存环境中运行，而内核应用程序则不是。在各自的环境中还存在的差异有，可用的特性，以及性能、占用空间和内存使用。

VxWorks内核应用程序以内核模式在内核内存空间中运行。它们可以与内核链接，或者在运行时下载到内核中（作为可下载内核模块DKMs）。因为内核应用程序在内核中运行，所以系统不能保护内核免受应用程序方面的不当行为，也不能保护应用程序彼此之间不发生不当行为。在运行时，VxWorks系统任务（如tExcTask）和应用程序任务仅通过它们的优先级和代码的功能来区分。

RTP应用程序在用户模式下作为实时进程执行，每个进程都在它们自己的受保护内存空间中，该内存空间与其他RTP应用程序和内核分开。当该保护是系统的关键考虑时，应使用RTP应用程序。RTP应用程序是独立于内核构建和存储的（在开发期间在主机上以及部署系统的目标上）。VxWorks RTP应用程序与UNIX和Linux等其他操作系统的应用程序类似，只是VxWorks实时进程是为实时系统设计的（有关这方面的更多信息，请参阅2.实时过程：RTPs）。

内核应用程序环境和RTP应用程序环境具有许多相同的特性，但是可用的特性方面也有所不同。例如，驱动程序只能安装在内核中，而PTE52一致性只能在RTP环境中实现。

虽然在内核环境或RTP环境中执行代码的速度没有内在的差异，但是它可能受到代码优化（尽管在RTP环境中影响较小）和系统调用（特定于RTP环境）的使用的影响。尤其使用系统调用（通常是这种情况）的RTP应用程序比类似的内核应用程序慢。此外，VSB项目可用于从各种VxWorks库中删除RTP处理代码（使用Real-Time Process选项），这提高了这些库的性能并减小了它们的大小。

此外，配置有RTP支持组件的VxWorks图像将略大于不支持RTX支持组件的VxWorks图像。而且，对于采用RTP应用程序的系统，总体内存需求通常比可比较的内核应用程序的系统大，因为RTP应用程序包括用户端库（它复制了内核中已经包括的库）。

1.2 C和C++库

VxWorks本地C库和Dinkum C和C++库都是为VxWorks应用程序开发提供的。

使用VxWorks本地C库和Dinkum库

VxWorks本地库用于C内核应用程序开发，Dinkum库在所有其他情况下使用。

VxWorks本机C库提供ANSI规范之外的功能。注意，它们不支持宽字符或多字节字符。

有关这些库的更多信息，请参见VxWorks和Dinkum API引用。有关C++设施的更多信息，请参见5 C++开发。

1.3 内核程序的结构

内核应用程序代码类似于普通的C或C++应用程序，除了它不需要传统的main函数（不像VxWorks基于进程的应用程序）。它只需要一个入口点函数，启动应用程序运行所需的所有任务。

程序入口点函数

入口点函数执行数据初始化，并启动运行中的应用程序使用的所有任务。

注意：如果内核应用程序包含一个main函数，则不要假定它将自动启动。下载或简单地存储在系统映像中的内核应用程序模块必须以交互方式启动（或者由已经运行的另一个应用程序启动）。操作系统还可以被配置为在引导时自动启动应用程序（参见使用userAppInit()自动运行应用程序的1.8）。

例如，内核应用程序可能有一个名为 MyAppStudio，它可能看起来像这样：

void myAppStartUp (void)

{

runFoo();

tidThis = taskSpawn("tThis", 200, 0, STACK\_SIZE,

(FUNCPTR) thisFunction,0,0,0,0,0,0,0,0,0,0);

tidThat = taskSpawn("tThat", 220, 0, STACK\_SIZE,

(FUNCPTR) thatFunction,0,0,0,0,0,0,0,0,0,0);

tidAnother = taskSpawn("tAnother", 230, 0, STACK\_SIZE,

(FUNCPTR) anotherFunction,0,0,0,0,0,0,0,0,0,0);

return (OK);

}

有关VxWorks任务和多任务的信息，请参见6, 多任务处理。有关使用C++的信息见5, C++开发。

加载内核程序和终止内核程序的回调函数

如果打算在运行时将内核应用程序模块下载到VxWorks，则可以通过实现\_VxModuleAutoInit()函数作为入口点，并使用带有loadModule.()的LOAD\_CALL\_INIT\_AUTO选项，让加载程序自动启动它。还可以使用LOAD\_CALL\_INIT\_MANUAL选项，以便在不自动调用初始化函数的情况下加载模块。类似地，可以使用UNLD\_CALL\_TERM\_AUTO选项的unldByModuleId()函数实现\_VxModuleAutoTerm()函数，该函数在卸载模块时自动调用。有关更多信息，请参见LoopLIB和UNLDLIB API。

1.4 VxWorks 头文件

许多内核应用程序使用VxWorks操作系统工具类或程序库。这需要源模块引用VxWorks头文件。

VxWorks系统头文件

VxWorks的头文件为所有的全局VxWorks函数提供ANSI C函数原型声明。VxWorks提供了ANSI X3.159—1989标准所指定的所有头文件。

VxWorks系统头文件被复制到VSB项目目录，位于VSBProjutdir/KrnL/h。

VxWorks的头文件：vxWorks.h

头文件 vxWorks.h 必须首先包含在使用VxWorks工具类的每个内核应用程序模块中。它包含许多其他VxWorks模块广泛使用的基本定义和类型。许多其他VxWorks头文件都需要这些定义。包含vxWorks.h，以下行：

#include <vxWorks.h>

VxWorks嵌套头文件

一些VxWorks设备使用其他的、较低级别的VxWorks工具类。例如，TTY管理工具使用环形缓冲库。TTY头文件TyLIb.h使用由环缓冲头文件RNGLYB.H提供的定义。

要求意识到头文件依赖性和排序，这是非常不方便的，所有VxWorks标头文件显式地包含所有必备的头文件。头文件tyLib.h本身包含头文件rngLib.h（基本的VxWorks头文件vxWorks.h，所有其他头文件都假定已经包含它）。

通常，显式地包含必备头文件会带来问题：头文件可能被包含多次并产生致命的编译错误（因为C预处理器将重复定义视为潜在的冲突源）。然而，所有VxWorks头文件都包含有条件的编译语句和定义，这些语句和定义确保它们的文本只包含一次，而不管include语句指定了多少次。因此，内核应用程序模块可以只包括它直接需要的那些头文件，而不考虑相互依赖或排序，并且不会出现冲突。

VxWorks私有头文件

VxWorks的一些元素是可能会改变的内部细节，因此不应在内核应用程序中引用。模块功能的唯一支持用途是通过头文件中的公共定义以及模块的功能接口。依附性确保应用程序代码不受VxWorks模块实现中内部更改的影响。

一些头文件使用HIDDEN注释标记：

/\* HIDDEN \*/

...

/\* END HIDDEN \*/

内部细节也隐藏在私有的头文件中，这些文件位于私有的目录中。私有头文件的名称与公共头文件具有相同的根，但加上P。例如，semLib的私有头文件是semLibP.h。

头文件和extern "C" 申明

不要在 extern "C" 语句括号内包含系统头文件。所有的VxWorks系统头文件都已经配置成与C和C++编译器一起使用。如果在整个头文件的内容上强制C链接，则可能会在VxWorks头中无效。以下是不正确的：

#ifdef \_cplusplus

extern "C" {

#endif

#include <stdio.h>

应该使用下面的写法：

#include <stdio.h>

#ifdef \_cplusplus

extern "C" {

#endif

另外，在C++源文件中不要把头文件包含在extern "C"申明语句中。以下是不正确的：

extern "C" {

#include <stdio.h>

}

应该使用下面的写法：

#include <stdio.h>

1.5 内核对象的静态实例化

VxWorks内核对象，如任务和信号量可以静态实例化，也可以动态实例化。静态实例化在性能和确定性方面提供了优势。VxWorks为静态实例化提供了特殊的宏。

内核对象的静态实例化意味着该对象被声明为编译时变量（使用特殊的VxWorks宏），其范围通常是全局的。因此，编译器为应用程序中的对象分配存储，并且不需要运行时内存分配。因此，在启动时，该对象立即可用于初始化。

与静态实例化不同，内核对象的动态实例化涉及系统内存的运行时分配，然后在可以使用之前初始化对象。对象删除又涉及对象的无效，然后将内存返回给系统。对象创建和删除都依赖于动态内存分配，通常使用malloc()和free()函数。

因此，使用动态实例化的应用程序必须考虑系统可能耗尽内存，并且不能创建对象（并且诉诸一些适当的错误恢复过程或中止）的可能性。此外，动态分配是一个相对缓慢的操作，可能会阻塞调用任务，并使系统性能不确定性。

与静态实例化不同，动态实例化可以通过单个函数调用（例如，taskSpawn()、semXCreate()等完成。

下面的代码示例说明了动态实例化与静态实例化之间的区别（但是不使用任何VxWorks对象实例化宏）。

动态实例化

struct my\_object \* pMyObj;

...

pMyObj = (struct my\_object \*) malloc (sizeof (struct my\_object));

if (pMyObj != NULL)

{

fooObjectInit (pMyOjb);

return (OK);

}

else

{

/\* failure path \*/

return (ERROR);

}

静态实例化

struct my\_object myObj;

...

fooObjectInit (&myOjb);

/\* myObj now ready for use \*/

可以被静态实例化的内核对象

内核任务

信号量

消息队列

看门狗定时器

有关详细信息，请参阅内核任务的静态实例化，p.91，内核信号量的静态实例化，p.118，内核消息队列的静态实例化，p.133，以及看门狗定时器的静态实例化，p.174。

静态实例化的代码大小

对象的编译时声明不占用可执行文件、VxWorks映像或存储介质（如闪存）中的任何空间。如果在编译时声明对象但未初始化，则编译器将其放置在未初始化数据（BSS）部分中。ANSI C标准要求未初始化的数据具有零值。

当未初始化的数据有助于运行时内存占用时，动态分配也是如此。在这两种情况下，内存占用都是相同的。

静态实例化的优点

内核对象的静态实例化比动态实例化提供了几个优点：

对象的静态实例化速度更快，确定性更强。

应用逻辑更简单，因为它不必考虑动态分配可能失败的可能性。

对象的静态声明不能失败，除非程序本身太大，无法适应系统内存。

程序和静态实例化

内核对象的静态实例化对实时应用程序具有显著的优势，大多数应用程序可以在一定程度上利用静态实例化。大多数应用程序要求某些对象在其执行的生命周期内可用，并且它们永远不会被删除。因此，这些对象可以静态实例化。尽可能使用静态实例化使应用程序更加健壮、确定性和快速。

注意：静态实例化只能够应该在开发内核程序，不能使用在开发RTP程序上。

静态声明的作用域

内核对象通常被声明为全局变量，因为对象ID通常用于任务间通信和同步。然而，它们不必是全局变量的。对象声明也可以在函数范围内完成，只要对象在其使用期间保持在有效范围内。

关于宏使用的注意事项

为了确保适当的宏扩展，如果任务实例化宏实现的声明或调用超过一行，则它们必须使用反斜杠字符。例如，以下确保编译器将正确地处理具有VX\_TASK\_INSTANTIATE宏的任务的静态实例化：

myTaskId = VX\_TASK\_INSTANTIATE(myTask, 100, 0, 4096, pEntry, \

0,1,2,3,4,5,6,7,8,9);

在内核任务的静态实例化p.91、内核信号量的静态实例化p.118、内核消息队列的静态实例化p.133和看门狗定时器的静态实例化p.174中详细描述了宏使用。

1.6 启动时间钩子函数

在引导过程中的特定点调用的VxWorks钩子函数可用于添加自己的引导时功能或测试代码。钩子函数允许您添加代码，而无需修改引导加载程序或操作系统代码本身，或者无需为自己的代码创建自定义组件。

您可以添加在操作系统完全启动和运行之前执行时间关键操作的代码，或覆盖系统默认特性的代码。例如，可以使用此钩子函数功能来执行以下操作：

提供设备启动的早期反馈（将消息打印到LCD显示器）。

在启动过程中重置看门狗定时器。

调用自定义诊断或测试代码。

启动初始化过程中不需要联网的应用程序。

读取硬件版本或测试设备状态的GPIO。

早期配置FPGA。

启动时间钩子函数通过在VxWorks配置中包括特定组件来启用。钩子函数必须从usrAppInit.c文件中调用（在VIP项目中）。

有关在VxWorks启动后使用usrAppInit.c代码的信息，请参阅1.8“使用userAppInit()自动运行应用程序”，第11页。

1.7 启动时间钩子函数组件

钩子函数 描述 组件

usrPreKernelAppInit() 在usrInit()开始时调用，恰好在cacheLibInit()之后调用。也就是说，在板子启动之后和内核初始化开始之前。INCLUDE\_USER\_PRE\_KERNEL\_APPL\_INIT

usrPostKernelAppInit() 在usrKernelCreateInit()之后调用，这是大部分的内核功能都有了，但是I/O和网络还没有。INCLUDE\_USER\_POST\_KERNEL\_APPL\_INIT

usrPreNetworkAppInit() 在网络初始化后调用。这个时候I/O和网络都可以了。INCLUDE\_USER\_PRE\_NETWORK\_APPL\_INIT

使用注意事项

回调函数必须从usrAppInit.c文件中调用（在VIP项目中）。

对于usrPreKernelAppInit()，不能使用VxWorks API。只使用自己的代码（例如，初始化寄存器或点亮一个LED）。

对于usrPostKernelAppInit()，可以使用大多数内核API（用于创建和管理任务、信号量、消息队列等）。例外情况是那些依赖于系统时钟的API，即不能使用具有超时参数的API。I/O系统和网络在这一点上不可用。

对于usrPreNetworkAppInit()，除了网络之外，所有内核API都可以使用。

1.8 利用usrAppInit()在系统启动的时候运行程序

第一步：增加组件INCLUDE\_USER\_APPL

第二步：在usrAppInit()，调用要启动程序的入口函数，如：

void usrAppInit (void)

{

#ifdef USER\_APPL\_INIT

USER\_APPL\_INIT; /\* for backwards compatibility \*/

#endif

myAppStartUp();

}

第三步：链接DKM项目作为VxWorks内核对象。