ANYKA
芯 随 你 动

版本: 1.0.0 2011 年 12 月

Sword37AKOS 子模块

接口说明

声　　明

联　系　方　式

安凯（广州）微电子技术有限公司

地址：广州科学城科学大道 182 号创新大厦 C1 区 3 楼
电话: (86)-20-3221 9000
传真: (86)-20-3221 9258
邮编: 510663

销售热线:

(86)-20-3221 9499

电子邮箱:

sales@anyka.com

主页:

http: //www. anyka.com

版 本 变 更 说 明

以下表格对于本文档的版本变更做一个简要的说明。版本变更仅限于技术内容的变更，不包括版式、格式、句法等的变更。

版本	说明	完成日期
V1.0.0	正式发布	2011 年 12 月

目录

1 AKOS简介

AKOS 多任务整体软件架构如图 1-1 所示。AKOS 整体软件架构主要包括 Thread 线程模块、App 应用程序模块、AppMgr 任务管理模块以及 AKMsgDispatch 消息管理模块等。

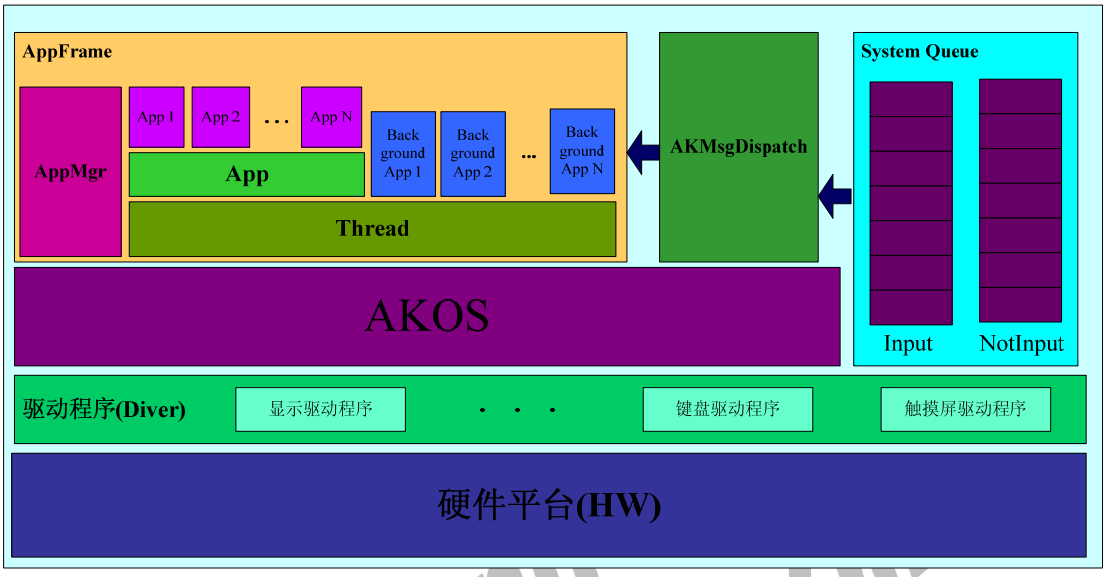

图 1-1 AKOS 整体软件架构图

1.1 AKThread

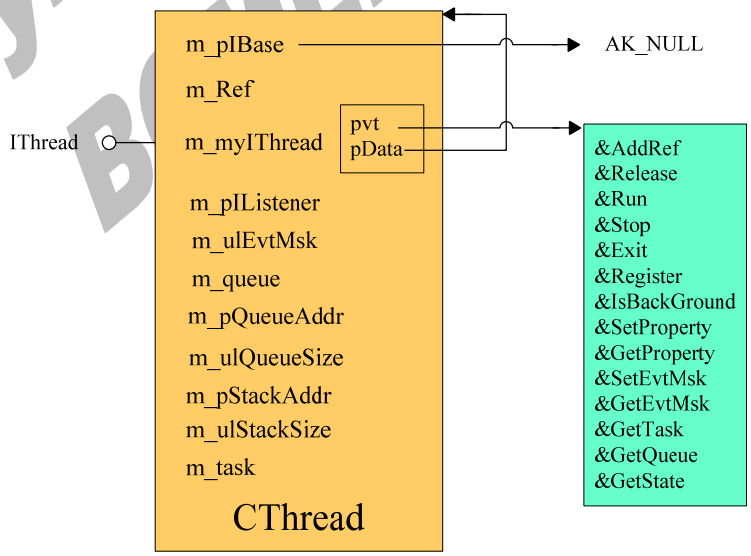

图 1-2 AKThread

1.2 AKApp

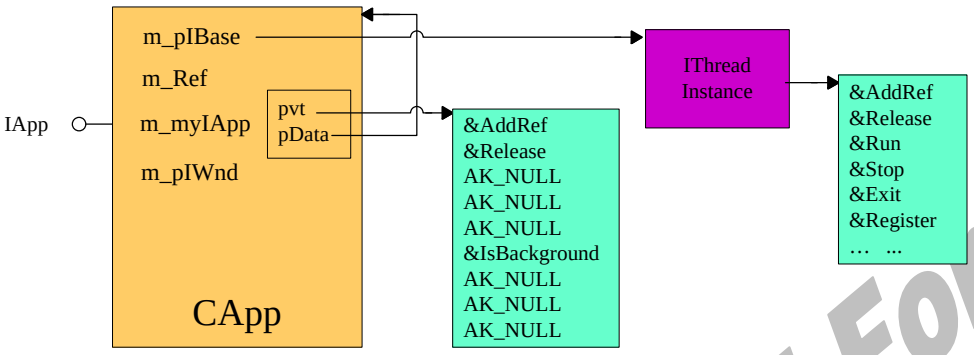

图 1-3 AKApp

1.3 AKAppMgr

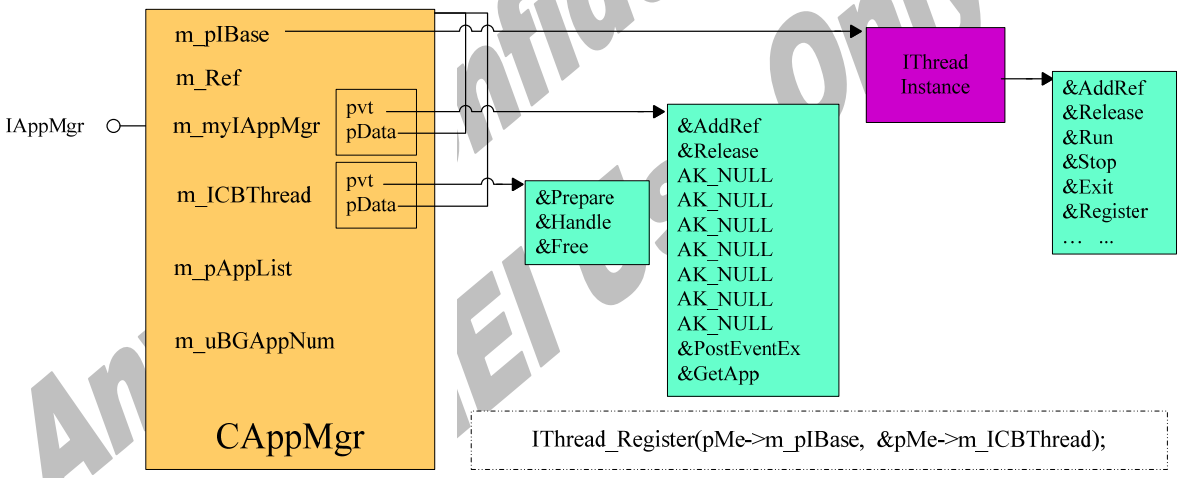

图 1-4 AKAppMgr

1.4 AKXxxApp

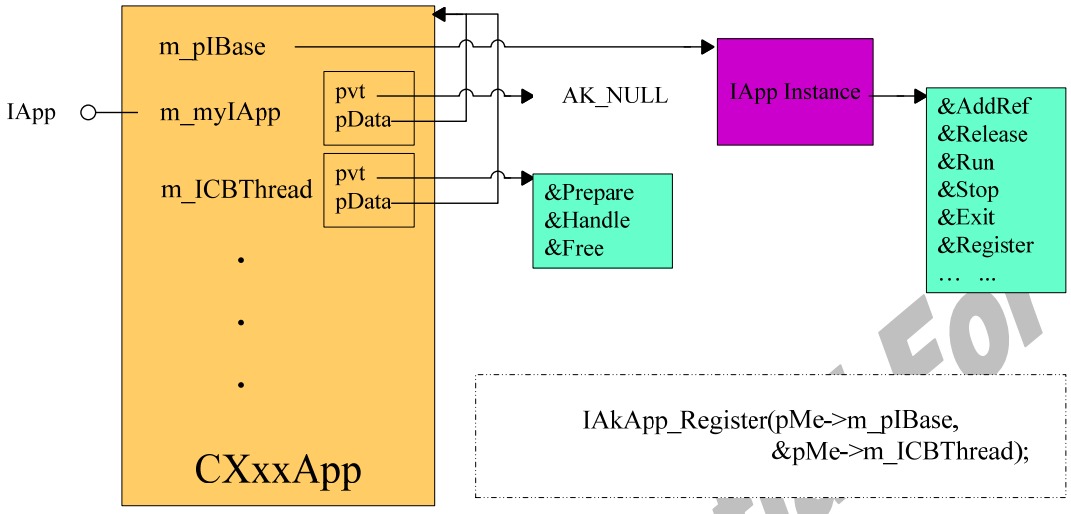

图 1-5 AKXxxApp

1.5 AKXxxBGApp

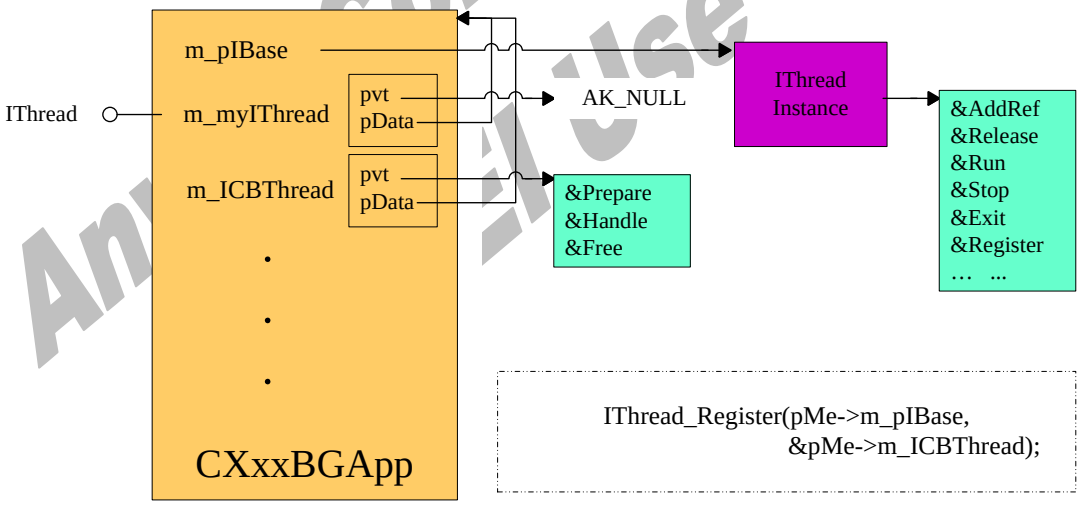

图 1-6 AKXxxBGApp

1.6 继承关系图

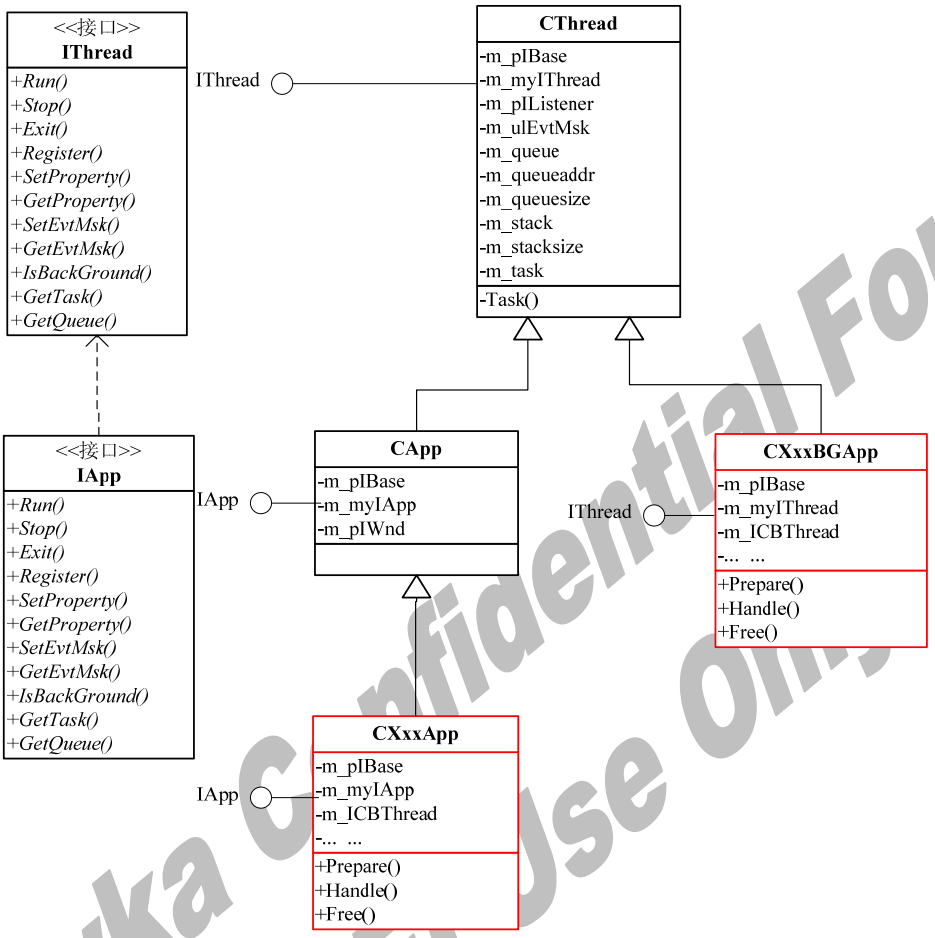

图 1-7 继承关系 1

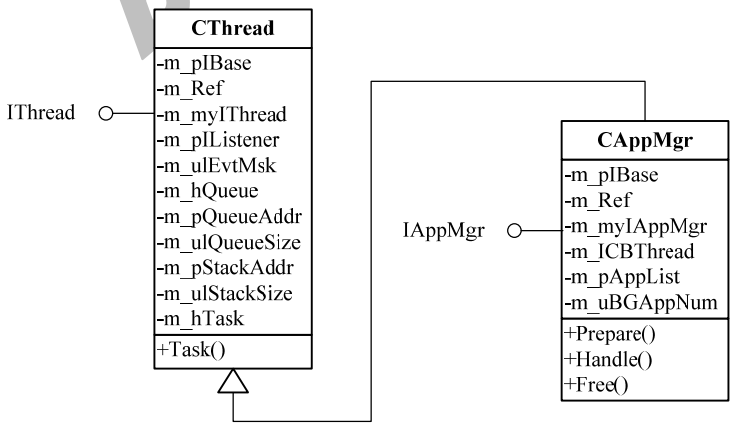

图 1-8 继承关系 2

2 Ithread

2.1 简介

IThread 为线程调度框架的封装，包括子线程管理，其原理框图如下图所示。IThread 接口说明见下一节。

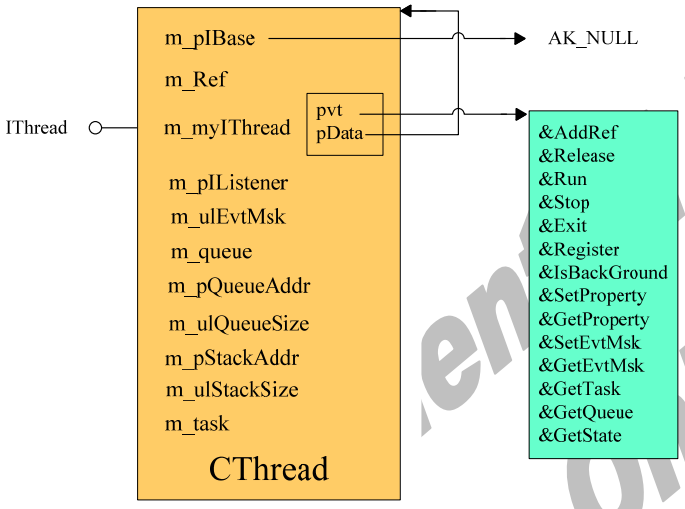

图 2-1 IThread 原理框图

2.2 接口说明

2.2.1 IThread_AddRef

原　　型	T_U32 IThread _AddRef(IThread *p IThread);	
功能概述	将该组件实例引用计数加 1。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IThread _AddRef(p IThread);	

2.2.2 IThread _Release

原　　型	T_U32 IThread _Release (IThread *p IThread);	
功能概述	将该组件实例引用计数减 1；如果引用计数为 0，则释放该组件实例。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IThread _Release (p IThread);	

2.2.3 IThread _Run

原　　型	T_S32 IThread _ Run (IThread *p IThread);	
功能概述	运行改线程，如果该线程处于 Terminate 状态，则先 Reset 该线程，再将它运行。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用完 IThread _ Run，该线程不一定立刻运行，除非没有比它高的线程和高级中断服务程序在运行。	
调用示例	IThread _ Run (p IThread);	

2.2.4 IThread _Terminate

原　　型	T_ S32 IThread _ Terminate (IThread *p IThread);	
功能概述	终止该线程运行，该线程将不再被 AKOS 调度，直到用户再次调用 IThread _ Run。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用该接口后，线程状态改为 Terminate 状态。	
调用示例	IThread _ Stop (p IThread);	

2.2.5 IThread _Exit

原　　型	T_ S32 IThread _ Exit (IThread *p IThread);	
功能概述	退出该线程，包括先终止该线程，然后从 AKOS 线程链表中删除该线程。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IThread _ Exit (p IThread);	

2.2.6 IThread _ IsBackground

原　　型	T_ S32 IThread _ IsBackground (IThread *p IThread);	
功能概述	返回当前应用是否是后台应用。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	AK_TRUE	表示该应用是后台应用
	AK_FALSE	表示该应用是前台应用
注意事项	继承 IThread 的应用是后台应用，继承 IApp 的应用是前台应用。	
调用示例	IThread _ IsBackground (p IThread);	

2.2.7 IThread _Register

原　　型	T_ S32 IThread _Register (IThread *p IThread， ICBThread *pIListener);	
功能概述	注册线程回调接口，这样应用的处理函数才有机会得到执行。	
参数说明	IThread　[in]	指向该组件实例的接口指针
	pIListener [in]	指向线程回调接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	每一个应用必须要进行注册。	
调用示例	IThread _Register (p IThread，pIListener);	

2.2.8 IThread _ SetProperty

原　　型	T_ S32 IThread _ SetProperty (IThread *p IThread，T_U16 wPropID, T_VOID* pPropData);	
功能概述	设置线程属性值，如事件掩码等。	
参数说明	IThread [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [in]	指向要设置的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IThread _ SetProperty (p IThread，wPropID，pPropData);	

2.2.9 IThread _ GetProperty

原　　型	T_ S32 IThread _ GetProperty (IThread *p IThread，T_U16 wPropID, T_VOID* pPropData);	
功能概述	获取线程属性值，如事件掩码等。	
参数说明	IThread [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [out]	指向要获取的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	T_U32 ulEvtMsk = 0x00000000; IThread _ GetProperty (p IThread，THREAD_PROP_EVENT_MSK，(T_VOID*)&ulEvtMsk);	

2.2.10 IThread _ SetEvtMsk

原　　型	T_ S32 IThread _ SetEvtMsk (IThread *p IThread，T_U32 ulEvtMsk);	
功能概述	设置线程关心的事件类掩码。	
参数说明	IThread [in]	指向该组件实例的接口指针
	ulEvtMsk [in]	事件掩码值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IThread _ SetEvtMsk (p IThread，0xffffffff);	

2.2.11 IThread _ GetEvtMsk

原　　型	T_ U32 IThread _ GetEvtMsk (IThread *p IThread);	
功能概述	获取线程关心的事件类掩码。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	表示线程关心的事件类掩码
注意事项		
调用示例	IThread _ GetEvtMsk (p IThread);	

2.2.12 IThread _GetState

原　　型	T_U16 IThread _GetState (IThread *p IThread);	
功能概述	获取线程当前状态。	
参数说明	IThread [in]	指向该组件实例的接口指针
返回值说明	T_U16 类型值	表示线程当前状态
注意事项	线程状态有： THREAD_STATE_READY、 THREAD_STATE_WAITING、 THREAD_STATE_SUSPEND、 THREAD_STATE_TERMINATE 等	
调用示例	IThread _GetState (p IThread);	

3 IAKApp

3.1 前台应用程序

前台应用程序包括一个头文件和一个源文件。头文件定义初始化参数类型和应用程序结构类型，以及应用程序创建函数声明。头文件如下所示。

```
/*****************************************************************************
*
* Copyrights (C) 2002, ANYKA software Inc
* All rights reserced.
*
* File name: AKXxxApp.h
* Function:
* Author:
* Date:
* Version: 1.0
*
*****************************************************************************/
#ifndef __AKXXXAPP_H__
#define __AKXXXAPP_H__
#include "AKApp.h"

////////////////////////////////////////////////////////////////////////////////////////////////////
typedef struct tagCXxxApp
{
    IApp      *m_pIBase;  //Base class instance...
    IApp       m_myIApp;
    ICBThread m_ICBThread;
    //Add your member here...

}CXxxApp;

////////////////////////////////////////////////////////////////////////////////////////////////////

T_S32   CXxxApp_New(IApp **ppi);

#endif //__AKXXXAPP_H__
```

注意：其中 XXX 将替换为具体的应用程序名；应用程序结构类型的第一个成员必须为它的基类指针，否则程序将会产生不可预期的错误。

源文件由三个回调函数、一个构造函数、一个析构函数和一个创建函数构成。在应用的构造函数里必须完成以下两项工作：

1）创建父类实例，负责任务的创建、销毁和消息的接收。

2）注册回调接口，当父类创建的任务开始运行时，它会调用回调函数 Prepare 让应用做好准备工作；当任务进入消息循环后，收到每一条消息都会调用回调函数 Handle 让应用去处理它们；在父类实例退出时，它会调用回调函数 Free 告知应用要退出，请做好善后处理。

回调接口的虚表含有三个函数指针，如果不想实现里面的某一个，可以直接将对应的指针赋为 AK_NULL，但 Hanle 是必须要实现的，否则该应用就没有意义，因为它不处理任何消息。

源文件如下所示：

```
* Version: 1.0
*
*****************************************************************************/
#include "AKXxxApp.h"
#include "eng_debug.h"
#include "Fwl_sysevent.h"
#include "AKAppmgr.h"

//#############################################################################################

//=============================================================================================
//        <---- Function Declaration ---->
//=============================================================================================
static T_S32 CXxxApp_ICBThread_Prepare(ICBThread *pICBThread);
static T_S32 CXxxApp_ICBThread_Handle(ICBThread *pICBThread, T_SYS_EVTID eEvent, T_SYS_PARAM *pEvtParam);
static T_S32 CXxxApp_ICBThread_Free(ICBThread *pICBThread);
/////////////////////////////////////////////////////////////////////////////////////////////
static const AK_VTABLE(ICBThread) g_ICBXxxAppFuncs =
{
    CXxxApp_ICBThread_Prepare,
    CXxxApp_ICBThread_Handle,
    CXxxApp_ICBThread_Free
};

/*=============================================================================================*/
/*****************************************************************************
* @BRIEF
* @AUTHOR
* @DATE   2007-09-05
* @PARAM
* @PARAM
* @RETURN T_S32 The result of this function executing.
* @RETVAL AK_SUCCESS      Execute successfully.
* @RETVAL Not AK_SUCCESS  The error type value.
*****************************************************************************/
static T_S32 CXxxApp_Constructor(CXxxApp *pMe)
{
    T_S32 lRet = AK_SUCCESS;
    T_AppInfo *pAppInfo = AK_NULL;
    T_APP_INITPARAM sInitParam;

    pAppInfo = AK_GetDefaultAppInfo(AKAPP_CLSID_XXX);
    if (AK_NULL == pAppInfo)
    {
        return AK_EUNSUPPORT;
    }
    sInitParam.base = pAppInfo->sInitparam;
    sInitParam.wnd.dispbuf      = AK_NULL;
    sInitParam.wnd.height       = Fwl_GetLcdHeight(LCD_0);
    sInitParam.wnd.width        = Fwl_GetLcdWidth(LCD_0);
    sInitParam.wnd.type         = AK_DISP_RGB;

    lRet = CApp_New(&pMe->m_pIBase, &sInitParam);
    lRet |= IAkApp_Register(pMe->m_pIBase,
                            &pMe->m_ICBThread);
    //Add your init code here...


    return lRet;
} ? end CXxxApp_Constructor ?
```

```
/*****************************************************************************
* @BRIEF
* @AUTHOR
* @DATE    2007-09-05
* @PARAM
* @PARAM
* @RETURN  T_S32  The result of this function executing.
* @RETVAL  AK_SUCCESS      Execute successfully.
* @RETVAL  Not AK_SUCCESS  The error type value.
*****************************************************************************/
static T_S32 CXxxApp_Destructor(CXxxApp *pMe)
{
    AK_RELEASEIF(pMe->m_pIBase);
    //Add your code here...


    //Add your code end.
    Fwl_Free(pMe);

    return AK_SUCCESS;
}
```

```
/*******************************************************************************
* @BRIEF
* @AUTHOR
* @DATE   2007-09-05
* @PARAM
* @PARAM
* @RETURN T_S32 The result of this function executing.
* @RETVAL AK_SUCCESS      Execute successfully.
* @RETVAL Not AK_SUCCESS The error type value.
*******************************************************************************/
static T_S32 CXxxApp_ICBThread_Prepare(ICBThread *pICBThread)
{
    CXxxApp *pMe = (CXxxApp *)pICBThread->pData;
    //Add your Prepare code here...

    return AK_SUCCESS;
}

/*******************************************************************************
* @BRIEF
* @AUTHOR
* @DATE   2007-09-05
* @PARAM
* @PARAM
* @RETURN T_S32 The result of this function executing.
* @RETVAL AK_SUCCESS      Execute successfully.
* @RETVAL Not AK_SUCCESS The error type value.
*******************************************************************************/
static T_S32 CXxxApp_ICBThread_Handle(ICBThread *pICBThread, T_SYS_EVTID eEvent, T_SYS_PARAM *pEvtParam)
{
    CXxxApp *pMe = (CXxxApp *)pICBThread->pData;
    //Add your Handle code here...

    return AK_SUCCESS;
}
/*******************************************************************************
* @BRIEF
* @AUTHOR
* @DATE   2007-09-05
* @PARAM
* @PARAM
* @RETURN T_S32 The result of this function executing.
* @RETVAL AK_SUCCESS      Execute successfully.
* @RETVAL Not AK_SUCCESS The error type value.
*******************************************************************************/
static T_S32 CXxxApp_ICBThread_Free(ICBThread *pICBThread)
{
    CXxxApp *pMe = (CXxxApp *)pICBThread->pData;

    CXxxApp_Destructor(pMe);

    return AK_SUCCESS;
}


/*******************************************************************************
* @BRIEF
* @AUTHOR
* @DATE   2007-09-05
* @PARAM
* @PARAM
* @RETURN T_S32 The result of this function executing.
* @RETVAL AK_SUCCESS      Execute successfully.
* @RETVAL Not AK_SUCCESS The error type value.
*******************************************************************************/
T_S32   CXxxApp_New(IApp **ppo)
{
    T_S32 nErr = AK_SUCCESS;
    CXxxApp *pNew = AK_NULL;
```

```
    if (AK_NULL == ppo)
    {
        return AK_EBADPARAM;
    }

    *ppo = AK_NULL;

    do
    {
        pNew = AK_MALLOCRECORD(CXxxApp);
        AK_BREAKIF(AK_NULL == pNew, nErr, AK_ENOMEMORY);

        pNew->m_myIApp.pvt    = AK_NULL;
        pNew->m_myIApp.pData = (T_VOID*)pNew;
        AK_SETVT(&(pNew->m_ICBThread), &g_ICBXxxAppFuncs);
        pNew->m_ICBThread.pData = (T_VOID*)pNew;

        nErr = CXxxApp_Constructor(pNew);
        if (AK_IS_FAILURE(nErr))
        {
            IAkApp_Register(pNew->m_pIBase,AK_NULL);
            CXxxApp_Destructor(pNew);
            pNew = AK_NULL;
            break;
        }

        *ppo = (IApp*)&pNew->m_myIApp;

    } ? end do ?  while(AK_FALSE);

    return nErr;
} ? end CXxxApp_New ?
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//              <----------          File    End     ---------->
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

其中下面是为虚表的每一个函数指针赋值，如某一个应用不需要 Prerare，那么就可以将第一个直接写上 AK_NULL 即可。如下所示。

```
static const AK_VTABLE(ICBThread) g_ICBXxxAppFuncs =
{
    CXxxApp_ICBThread_Prepare,
    CXxxApp_ICBThread_Handle,
    CXxxApp_ICBThread_Free
};
```

3.1.1 应用程序创建

创建一个应用在 CXxxApp_New 中完成，步骤如下：

1） 首先在 CXxxApp_New 中分配一块空间，用于存放应用结构体内容。

2） 初始化虚表和接口中的 pData 指针指向刚分配空间的首地址，紧接着初始化回调接口虚表指针和 pData。

3） 调用应用的构造函数用于初始化一下变量，在构造函数里完成了父类实例的创建和回调接口的注册。

4） 最后返回本应用接口地址，这样通过 CXxxApp_New 就可以获取该应用的接口指针。其中 param 为输入参数，含有要创建应用的优先级、时间片、堆栈大小和队列大小。父实例负责任务的创建、销毁和接收到消息后调用回调函数 Handle 让本应用处理。

3.1.2 应用程序执行时序

通过 CXxxApp_New 创建一个任务后，并不会直接运行，需要调用 IAkApp_Run(pIApp)来运行该任务。由于现在所有的前后台应用都要由 AppMgr 来管理，所有要通过如下语句将应用交给 AppMgr 管理：

```
T_AppEntry AppEntry;

AppEntry.wAppCls   = AKAPP_CLSID_MMI;
AppEntry.pIThread = (IThread*)pIApp;
IAppMgr_AddEntry(pIAppMgr, &AppEntry, AK_TRUE);
```

其中，wAppCls 为应用的 Class ID；pIThread 为应用的接口指针。

通过 IAppMgr_AddEntry(pIAppMgr, &AppEntry, AK_TRUE);可以将应用添加到应用列表里，最后一个参数表示该应用要不要成为激活应用。（激活应用：处于最顶层的前台应用，只有激活应用才可以收到键盘、触摸屏等输入事件和绘制屏幕）IAppMgr_AddEntry 的另外一个功能是运行应用，所以在 CXxxApp_New 后不需要调用 IAkApp_Run 来运行应用，直接调用 IAppMgr_AddEntry 即可。

下面介绍应用运行之后如何调用回调接口的那三个函数：

1）在任务运行之后第一个调用的是 Prepare 函数，该函数提供一个在进入消息循环前让应用作准备的机会；

2）然后进入消息循环，每收到一条消息都会调用 Hanle 让应用处理；

3）在任务退出时，调用 Free 让应用释放一些资源。

3.2 后台应用程序

后台应用程序跟前台应用程序非常相似，形式也基本一样。从外表看，就是将 IApp 替换成了 IThread，其它没有什么区别。从继承图上可以看出，后台应用是直接继承 IThread 的，前台应用是继承 IApp 的，而 IApp 是继承 IThread 的，它们关键的区别就是父类不同。IApp 的功能通过查看 CApp 的定义就知道了，含有一个成员 IWnd，为一个窗口类。这是它们最主要的区别。后台应用的相关内容直接看模板代码就可以了，因为跟前台基本一样。

3.3 接口说明

IAKApp 应用程序模块主要是完成对线程 CWnd 的封装，响应输入消息、可被激活。其原理框图如图 3-1 所示。具体接口说明见下一节。

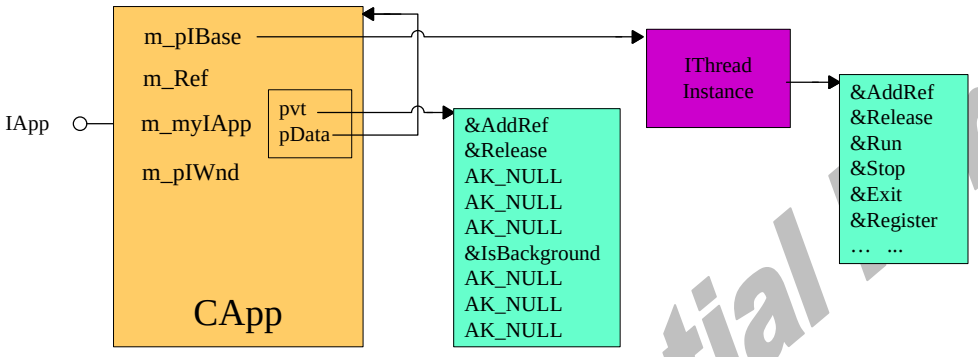

图 3-1 IAKApp 接口

3.3.1 IAkApp_AddRef

原　　型	T_U32 IAkApp_AddRef(IAkApp*pIAkApp);	
功能概述	将该应用实例引用计数加 1。	
参数说明	IAkApp　[in]	指向该应用实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IAkApp_AddRef(p IAkApp);	

3.3.2 IAkApp_Release

原　　型	T_U32 IAkApp_Release (IAkApp*p IAkApp);	
功能概述	将该应用实例引用计数减 1；如果引用计数为 0，则释放该组件实例。	
参数说明	IAkApp　[in]	指向该应用实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IAkApp_Release (p IAkApp);	

3.3.3 IAkApp_Run

原　　型	T_S32 IAkApp_ Run (IAkApp*p IAkApp);	
功能概述	运行改应用，如果该应用处于 Terminate 状态，则先 Reset 该线程，再将它运行。	
参数说明	IAkApp [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用完 IAkApp_ Run，该应用不一定立刻运行，除非没有比它高的线程和高级中断服务程序在运行。	
调用示例	IAkApp_ Run (p IAkApp);	

3.3.4 IAkApp_Terminate

原　　型	T_ S32 IAkApp_ Terminate (IAkApp*p IAkApp);	
功能概述	终止该应用运行，该线程将不再被 AKOS 调度，直到用户再次调用 IAkApp _ Run。	
参数说明	IAkApp [in]	指向该应用实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用该接口后，应用状态改为 Terminate 状态。	
调用示例	IAkApp_ Stop (p IAkApp);	

3.3.5 IAkApp_Exit

原　　型	T_ S32 IAkApp_ Exit (IAkApp*p IAkApp);	
功能概述	退出该应用，包括先终止该应用，然后从 AKOS 线程链表中删除该应用。	
参数说明	IAkApp[in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAkApp_ Exit (p IAkApp);	

3.3.6 IAkApp_Register

原　　型	T_ S32 IAkApp_Register (IAkApp*p IAkApp， ICBThread *pIListener);	
功能概述	注册应用回调接口，这样应用的处理函数才有机会得到执行。	
参数说明	IAkApp　[in]	指向该组件实例的接口指针
	pIListener [in]	指向线程回调接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	每一个应用必须要进行注册。	
调用示例	IAkApp_Register (p IAkApp，pIListener);	

3.3.7 IAkApp_ SetProperty

原　　型	T_ S32 IAkApp_ SetProperty (IAkApp*p IThread，T_U16 wPropID, T_VOID* pPropData);	
功能概述	设置应用属性值，如事件掩码等。	
参数说明	IAkApp [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [in]	指向要设置的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAkApp_ SetProperty (p IAkApp，wPropID，pPropData);	

3.3.8 IAkApp_ GetProperty

原　　型	T_ S32 IAkApp_ GetProperty (IAkApp*p IAkApp，T_U16 wPropID, T_VOID* pPropData);	
功能概述	获取应用属性值，如事件掩码等。	
参数说明	IAkApp [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [out]	指向要获取的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	T_U32 ulEvtMsk = 0x00000000; IAkApp_ GetProperty (p IAkApp，THREAD_PROP_EVENT_MSK，(T_VOID*)&ulEvtMsk);	

3.3.9 IAkApp_ SetEvtMsk

原　　型	T_ S32 IAkApp_ SetEvtMsk (IAkApp*p IAkApp，T_U32 ulEvtMsk);	
功能概述	设置应用关心的事件类掩码。	
参数说明	IAkApp　　[in]	指向该组件实例的接口指针
	ulEvtMsk　　[in]	事件掩码值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAkApp_ SetEvtMsk (p IAkApp，0xffffffff);	

3.3.10 IAkApp_ GetEvtMsk

原　　型	T_ U32 IAkApp_ GetEvtMsk (IAkApp*p IAkApp);	
功能概述	获取应用关心的事件类掩码。	
参数说明	IAkApp[in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	表示线程关心的事件类掩码
注意事项		
调用示例	IAkApp_ GetEvtMsk (p IAkApp);	

3.3.11 IAkApp_ GetIWnd

原　　型	IWnd * IAkApp_ GetIWnd (IAkApp*p IAkApp);	
功能概述	获取应用当前视窗接口指针。	
参数说明	IAkApp[in]	指向该组件实例的接口指针
返回值说明	IWnd*	应用内部视窗接口指针
注意事项		
调用示例	IWnd *pIWnd = AK_NULL; pIWnd = IAkApp_ GetIWnd (p IAkApp);	

4 IAppMgr

4.1 简介

IAppMgr 为应用程序管理框架，负责管理、查询、增减、激活各类 APP 等，通过消息驱动各类操作。应用管理模块应用框图如图 4-1 所示。详细接口说明见下一节。

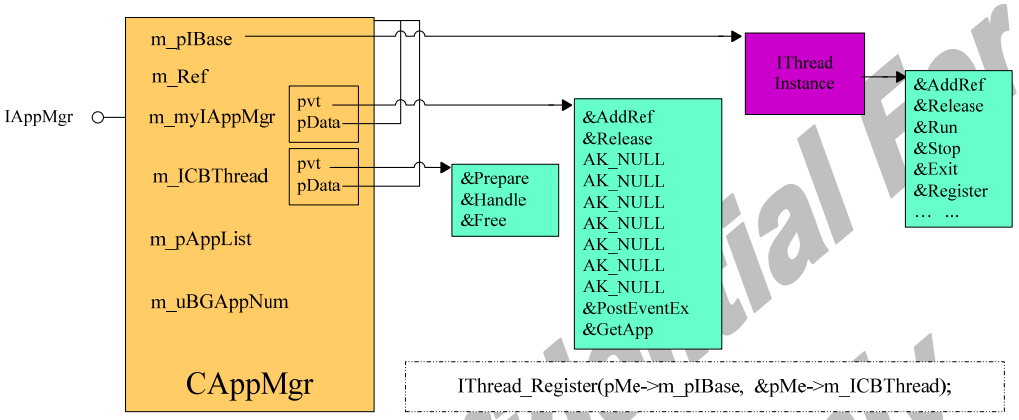

图 4-1 IAppMgr

4.2 接口说明

4.2.1 IAppMgr_AddRef

原　　型	T_U32 IAppMgr_AddRef(IAppMgr*p IAppMgr);	
功能概述	将该实例引用计数加 1。	
参数说明	IAkApp　[in]	指向该实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IAppMgr_AddRef(p IAppMgr);	

4.2.2 IAppMgr_Release

原　　型	T_U32 IAppMgr_Release (IAppMgr*p IAppMgr);	
功能概述	将该实例引用计数减 1；如果引用计数为 0，则释放该组件实例。	
参数说明	IAkApp　[in]	指向该实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IAppMgr_Release (p IAppMgr);	

4.2.3 IAppMgr _ GetActiveApp

原　　型	IThread * IAppMgr _ GetActiveApp (IAppMgr *p IAppMgr);	
功能概述	获取应用当前视窗接口指针。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
返回值说明	IThread*	返回激活应用接口指针
注意事项	可能会返回 AK_NULL。	
调用示例	IThread *pIThread = AK_NULL; pIThread = IAppMgr _ GetActiveApp (p IAppMgr);	

4.2.4 IAppMgr _ AddEntry

原　　型	T_ S32 IAppMgr _ AddEntry (IAppMgr *pIAppMgr， T_AppEntry *pAppEntry, T_BOOL bIsActive);	
功能概述	添加一个应用到应用管理器，如果成功添加，该应用将被运行。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	pAppEntry [in]	指向应用程序实例
	bIsActive [in]	要不要让该应用成为激活应用。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	参数 bIsActive 对后台应用无效。	
调用示例	T_AppEntry AppEntry; IApp *pIApp = AK_NULL; CMMI_New(&pIApp); AppEntry.wAppCls = AKAPP_CLSID_MMI; AppEntry.pIThread = (IThread*)pIApp; IAppMgr _ AddEntry (p IAppMgr，&AppEntry, AK_TRUE);	

4.2.5 IAppMgr _ DeleteEntry

原　　型	T_ S32 IAppMgr _ DeleteEntry (IAppMgr *pIAppMgr，T_U16 wAppCls);	
功能概述	从应用管理器中删除一个应用，并将该应用终止，删除。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	wAppCls [in]	应用的 Class ID。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAppMgr _ DeleteEntry (p IAppMgr，AKAPP_CLSID_MMI);	

4.2.6 IAppMgr _ ActiveApp

原　　型	T_ S32 IAppMgr _ ActiveApp (IAppMgr *pIAppMgr，T_U16 wAppCls);	
功能概述	激活一个前台应用，键盘、触摸屏等输入事件将发送给它。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	wAppCls [in]	应用的 Class ID。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAppMgr _ ActiveApp (p IAppMgr，AKAPP_CLSID_MMI);	

4.2.7 IAppMgr _ DeactiveApp

原　　型	T_ S32 IAppMgr _ DeactiveApp (IAppMgr *pIAppMgr，T_U16 wAppCls);	
功能概述	去激活一个前台应用，键盘、触摸屏等输入事件将不再发送给它，而发给新的激活应用。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	wAppCls [in]	应用的 Class ID。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAppMgr _ DeactiveApp (p IAppMgr，AKAPP_CLSID_MMI);	

4.2.8 IAppMgr _ PostEvent

原　　型	T_ S32 IAppMgr _ PostEvent (IAppMgr *pIAppMgr，T_U16 wAppCls, T_SYS_MAILBOX *pMailBox);	
功能概述	添加一个应用到应用管理器，如果成功添加，该应用将被运行。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	wAppCls [in]	要发给应用的 Class ID
	pMailBox [in]	指向消息结构体的指针。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	默认放在应用队列尾，如果要发给应用的队列满，发送方应用将被挂起。	
调用示例	IAppMgr _ PostEvent (p IAppMgr，AKAPP_CLSID_MMI, &mailbox);	

4.2.9 IAppMgr _ PostEventEx

原　　型	T_ S32 IAppMgr _ PostEventEx (IAppMgr *pIAppMgr，T_U16 wAppCls, T_SYS_MAILBOX *pMailBox, T_BOOL bIsHead, T_BOOL bIsSuspend);	
功能概述	添加一个应用到应用管理器，如果成功添加，该应用将被运行。	
参数说明	IAppMgr [in]	指向该组件实例的接口指针
	wAppCls [in]	要发给应用的 Class ID
	pMailBox [in]	指向消息结构体的指针。
	bIsHead [in]	要不要放在消息队列头
	bIsSuspend [in]	如果要发给应用的队列满，发送方要不要挂起。
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IAppMgr _ PostEventEx (pIAppMgr，AKAPP_CLSID_MMI, &mailbox, AK_TRUE, AK_FALSE);	

5 AKMsgDispatch

5.1 事件分类

5.1.1 系统事件

系统事件分为输入系统事件和非输入系统事件。输入系统事件主要是按键、触摸屏等终端输入的事件；非输入系统事件是除了输入系统事件外的其它系统事件，如：SYS_EVT_USB_PLUGIN 等。

系统事件的取值范围是 0xffff0000 ~ 0xffffffff，这主要是为了区别每个应用的内部事件。

应用的内部事件取值范围是 0x00000000 ~ 0x0000ffff。

系统事件都是从系统队列被派发给每个应用的。系统队列一共两个，一个用于存放键盘、触摸屏等输入事件的系统队列；另一个用于存放非输入系统事件的系统队列。通过如下的接口向系统队列发送事件：

1）输入系统队列：

T_S32 AK_PostTerminalInputEvent(T_SYS_MAILBOX *pMailBox);

T_S32 AK_PostTerminalInputEventEx(T_SYS_MAILBOX *pMailBox, T_pfnEvtCmp pfnCmp,

T_BOOL bIsUnique, T_BOOL bIsHead);

2）非输入系统队列：

T_S32 AK_PostEvent(T_SYS_MAILBOX *pMailBox);

T_S32 AK_PostEventToHead(T_SYS_MAILBOX *pMailBox);

T_S32 AK_PostUniqueEvent (T_SYS_MAILBOX *pMailBox, T_pfnEvtCmp pfnCmp);

T_S32 AK_PostUniqueEventToHead (T_SYS_MAILBOX *pMailBox, T_pfnEvtCmp pfnCmp);

5.1.1.1.1 输入系统事件

键盘输入事件：

SYS_EVT_USER_KEY。

触摸屏输入事件：

SYS_EVT_TSCR。

5.1.1.1.2 非输入系统事件

公共系统事件：

SYS_EVT_TIMER。

音频相关事件：

SYS_EVT_MEDIA、SYS_EVT_SDCB_MESSAGE。

MMI 相关事件：

SYS_EVT_PUB_TIMER、SYS_EVT_PINIO、SYS_EVT_SD_PLUG、SYS_EVT_USB_PLUGIN、… …

5.1.1.1.3 系统事件分类

由于系统事件较多，但不是每一个应用都关心这些系统事件，所以必须要有一种机制让每个应用只接收到它们关心的事件。为了实现这个目的，目前 AKOS 平台给出了下面的解决方案：

1）分类

将系统事件分类：目前最多支持分为 32 大类。每一类对应于一个 32 位整型数的一位。每个应用只要将自己所关心的事件大类对应的位置 1，不关心位全置 0，这样就可以将系统事件发个有需要的应用了。对于输入时间类事件，不论该应用是否设置了该类事件所对应的位，都会只发给激活应用。

目前事件的分类如下：

```
enum
{
    SYS_EVT_MSK_NONE    = 0x00000000,
    SYS_EVT_COMM_MSK    = 0x00000001,
    SYS_EVT_APP_MSK     = 0x00000002,
    SYS_EVT_INPUT_MSK   = 0x00000004,
    SYS_EVT_VATC_MSK    = 0x00000008,
    SYS_EVT_AUDIO_MSK   = 0x00000010,
    SYS_EVT_MMI_MSK     = 0x00000020,
    SYS_EVT_XXX_MSK     = 0x00000040
};
```

注：其中 SYS_EVT_XXX_MSK 为即将添加的分类，XXX 为分类名。

2）掩码

每一个应用含有一个 32 位整型的变量 m_ulEvtMsk 用于保存它所关心事件的掩码。在事件分发器分发系统事件时会根据每一个应用所设置的事件掩码来决定是否要将该事件发给该应用。应用通过如下接口设置应用的事件掩码：

前台应用：IAkApp_SetEvtMsk(pIApp, SYS_EVT_COMM_MSK | SYS_EVT_MMI_MSK);

后台应用：IThread_SetEvtMsk(pIThread, SYS_EVT_VATC_MSK);

5.1.2 应用程序事件

应用的内部事件取值范围是 0x00000000 ~ 0x0000ffff 之间。这些事件不会经过系统队列转发，只应该发送给该应用。应用程序内部事件的含义只有应用程序自己知道，其它应用并不清楚，所以每个应用可以定义值相等事件。它们的事件取值范围都是 0x00000000 ~ 0x0000ffff 之间。

向一个具体应用发送事件，用如下接口：

```
//默认放在队列尾，且如果队列满发送事件的应用自动挂起。
T_S32 IAppMgr_PostEvent (IAppMgr* pIAppMgr, T_U16 wAppCls, T_SYS_MAILBOX *pMailBox);
//扩展发送事件接口。
T_S32 IAppMgr_PostEventEx (IAppMgr* pIAppMgr, T_U16 wAppCls, T_SYS_MAILBOX *pMailBox,
T_BOOL bIsHead, T_BOOL bIsSuspend);
```

注：参数 pIAppMgr 为应用管理器接口指针，wAppCls 为要发给的应用 Class ID。

5.1.3 事件结构体定义

事件类型定义：

```
typedef T_U32  T_SYS_EVTID;
```

参数类型定义：

```
typedef union
{
  struct
  {
    T_U8 Param1;
    T_U8 Param2;
    T_U8 Param3;
    T_U8 Param4;
    T_U8 Param5;
    T_U8 Param6;
    T_U8 Param7;
    T_U8 Param8;
  } c;
  struct
  {
    T_U16 Param1;
    T_U16 Param2;
    T_U16 Param3;
    T_U16 Param4;
} s;
struct
  {
    T_U32 Param1;
    T_U32 Param2;
  } w;
  T_pVOID   lpParam;
  T_U32     lParam;
} T_SYS_PARAM;        ///< Union for system event parameter
```

消息结构体定义：

```
typedef struct
```

```
{
    /** event code*/
    T_SYS_EVTID    event;
    /** event parameter*/
    T_SYS_PARAM  param;
  T_BOOL          bIsUnique;   //No need to set value.
  T_BOOL          bIsHead;     //No need to set value.
  T_pfnEvtCmp     fnCmp;       //No need to set value.
}T_SYS_MAILBOX;
```

其中最后三个成员是不用赋值的，它是由系统内部维护的。开发人员只要关心 event 和 param。

5.1.4 常用事件示例说明

1）按键事件

M_EVT_USER_KEY（由 SYS_EVT_USER_KEY 事件经过 CMMI_TranslateEvt 接口转换而来），其参数 pEventParm->c.Param1 的值为按键的 keyID，例如 kbOK（即 OK 键）等，取值参见 T_eKEY_ID 枚举；参数 pEventParm->c.Param2 的值为按键的 pressType，例如 PRESS_SHORT（短按）、PRESS_LONG（长按）等，取值参见 T_PRESS_TYPE 枚举。

2）触屏示例

M_EVT_TOUCH_SCREEN（由 SYS_EVT_TSCR 事件经过 CMMI_TranslateEvt 接口转换而来），其参数 pEventParm->s.Param1 的值为触摸状态，例如 eTOUCHSCR_UP、eTOUCHSCR_DOWN 等，取值参见 T_TOUCHSCR_ACTION 枚举；其参数 pEventParm->s.Param2 的值为触摸点的坐标 x 值；其参数 pEventParm->s.Param3 的值为触摸点的坐标 y 值。

3）Timer 事件

VME_EVT_TIMER（由 SYS_EVT_TIMER 事件经过 CMMI_TranslateEvt 接口转换而来），其参数 pEventParm->w.Param1 的值为该 timer 的 id，即启动该定时器返回的。

5.2 消息传递机制

5.2.1 消息传递机制框图

AKMsgDispatch 为一个 HISR（高级中断）模块，负责消息的管理、分发和调度。如下图所示为消息传递机制框图。

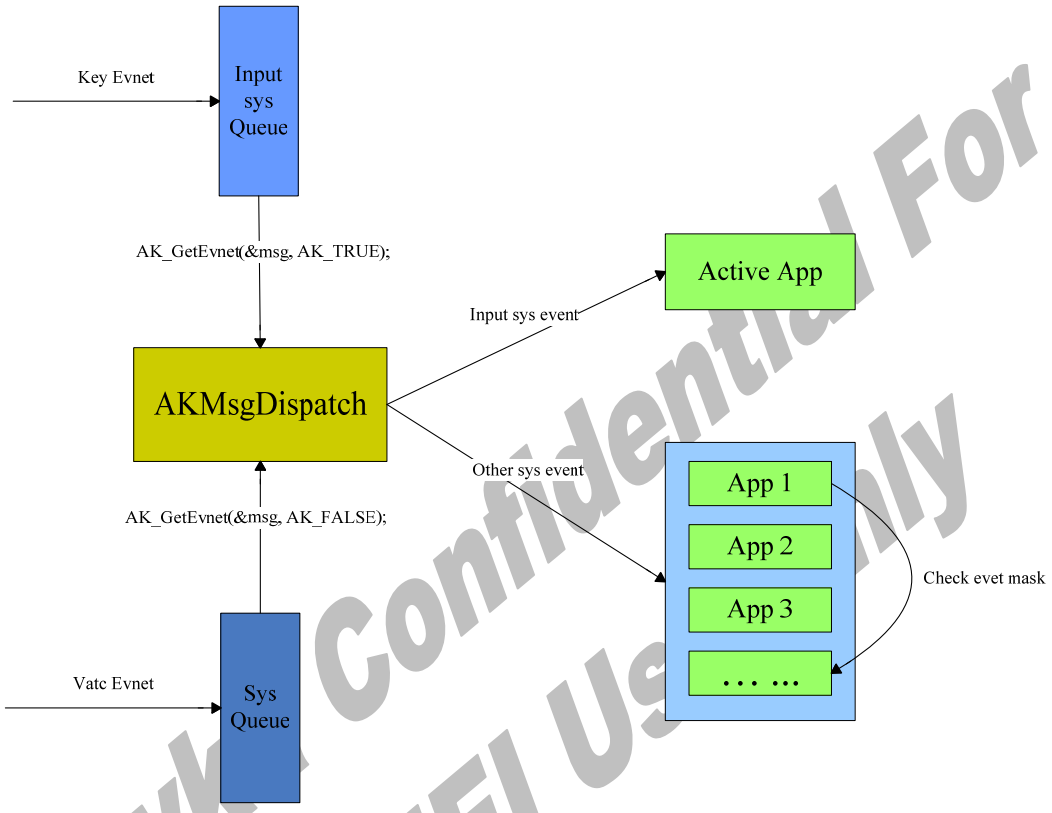

图 5-1 AKMsgDispatch 消息传递机制

AKMsgDispatch 的静态类图如下图所示。

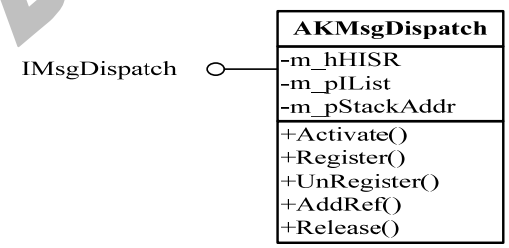

图 5-2 AKMsgDispatch 静态类图

5.2.2 AKMsgDispatch接口说明

5.2.2.1 IMsgDispatch_AddRef

原　　型	T_U32 IMsgDispatch_AddRef(IMsgDispatch *pIMsgDispatch);	
功能概述	将该组件实例引用计数加 1。	
参数说明	pIMsgDispatch [in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IMsgDispatch_AddRef(pIMsgDispatch);	

5.2.2.2 IMsgDispatch_Release

原　　型	T_U32 IMsgDispatch_Release (IMsgDispatch *pIMsgDispatch);	
功能概述	将该组件实例引用计数减 1；如果引用计数为 0，则释放该组件实例。	
参数说明	pIMsgDispatch [in]	指向该组件实例的接口指针
返回值说明	T_U32 类型值	返回当前该实例的引用计数
注意事项		
调用示例	IMsgDispatch_Release (pIMsgDispatch);	

5.2.2.3 IMsgDispatch_Activate

原　　型	T_U32 IMsgDispatch_Activate (IMsgDispatch *pIMsgDispatch);	
功能概述	用于触发 AKMsgDispatch 进行一次消息派发。	
参数说明	pIMsgDispatch [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IMsgDispatch_Activate(pIMsgDispatch);	

5.2.2.4 IMsgDispatch_Register

原　　型	T_U32 IMsgDispatch_Register (IMsgDispatch *pIMsgDispatch， IThread *pIThread);	
功能概述	将传入的应用接口指针注册到 AKMsgDispatch 的应用队列，这样系统消息才能派发给它，当然还需要它设置了该事件类掩码。	
参数说明	pIMsgDispatch [in]	指向该组件实例的接口指针
	pIThread [in]	指向应用程序实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	需要接收系统事件的应用必须调用该接口将自己添加到 AKMsgDispatch 的应用队列。	
调用示例	IMsgDispatch_Register (pIMsgDispatch，pIThread);	

5.2.2.5 IMsgDispatch_UnRegister

原　　型	T_U32 IMsgDispatch_ UnRegister (IMsgDispatch *pIMsgDispatch， IThread *pIThread);	
功能概述	取消注册到 AKMsgDispatch 应用队列的线程，这样系统消息不再会派发给它。	
参数说明	pIMsgDispatch [in]	指向该组件实例的接口指针
	pIThread [in]	指向应用程序实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	IMsgDispatch_ UnRegister (pIMsgDispatch，pIThread);	

5.2.3 AKMsgDispatch消息派发流程图

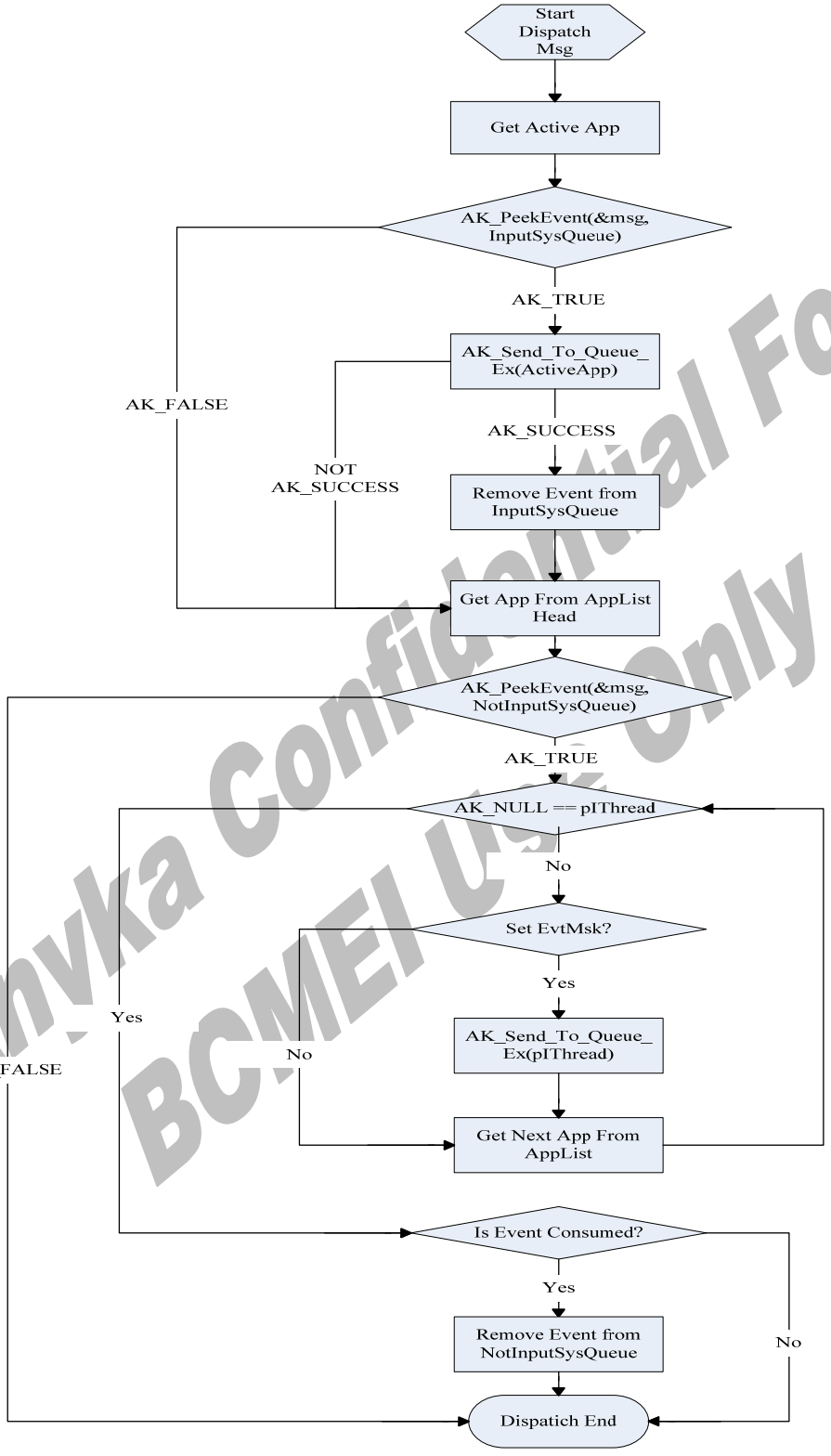

图 5-3 AKMsgDispatch 消息派发流程图

6 Subthread

6.1 工作原理

子线程是对 AKOS 任务的一种封装。子线程跟线程的概念基本一致，但不会默认创建消息队列。调用者需要提供子线程的优先级、时间片、主线程 ID、堆栈大小、执行体和 Abort 函数的入口地址等。根据这些信息子线程会创建一个任务，并将该子线程加入主线程的子线程列表中。当任务执行时就会调用传入的执行体函数，执行完后会向主线程发送一个完成消息告知主线程可以销毁自己。收到这个消息，主线程会从子线程列表中删除该子线程，并销毁该子线程中的任务。

如果子线程执行时间很长，用户可以通过 ISubThread_SetNotifyTimer 设置，让子线程定时告知主线程当前进度，以便显示当前进度条位置。通知的方式是向主线程发送一个事件 SYS_EVT_SUBTHREAD_NOTIFY，并将子线程控制块指针作为参数一起发送。如果子线程需要在执行完之前就终止它，用户则可以直接 Terminate 该子线程。但是不推荐使用这种粗暴的行为，这可能会带来一些隐患，如正在解析一张大图片时突然被 Terminate 可能会导致图象库没有释放一些临时打开和申请的资源，在下一次使用图象库时系统可能会崩溃。当然在 Terminate 一个子线程时会调用创建子线程时传入的 Abort 函数，会做一些释放资源的动作。但可能不能够将用到的库里面临时打开的文件给关闭掉等。使用该功能时用户需要谨慎操作。

6.2 应用框图

Subthread 子线程封装，可以自动完成子线程的构建、销毁、附加、剥离等子线程特性相关动作，其应用框图如下图所示。

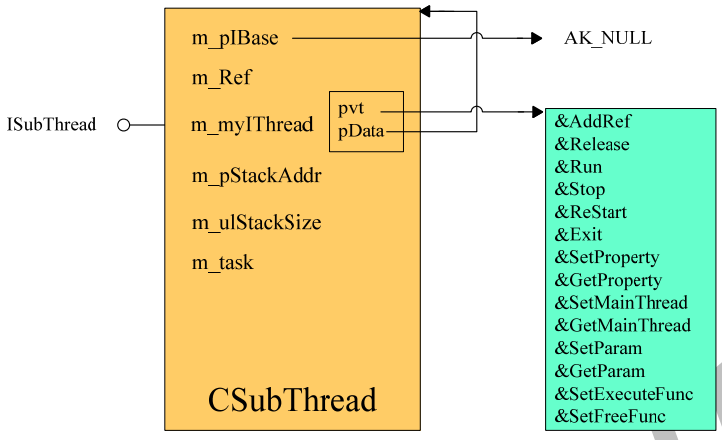

图 6-1 Subthread 子线程

6.3 接口说明

6.3.1 ISubThread _Resume

原　　型	T_S32 ISubThread _ Resume (ISubThread *p ISubThread);	
功能概述	运行改线程，如果该线程处于 Terminate 状态，则先 Reset 该线程，再将它运行。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用完 ISubThread _ Resume，该线程不一定立刻运行，除非没有比它高的线程和高级中断服务程序在运行。	
调用示例	ISubThread _ Resume (p ISubThread);	

6.3.2 ISubThread _Suspend

原　　型	T_ S32 ISubThread _ Suspend (ISubThread *p ISubThread);	
功能概述	挂起该线程，该线程将不再被 AKOS 调度，直到用户再次调用 ISubThread _Resume。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用该接口后，线程状态改为 Suspend 状态。	
调用示例	ISubThread _ Suspend (p ISubThread);	

6.3.3 ISubThread _Terminate

原　　型	T_ S32 ISubThread _ Terminate (ISubThread *p ISubThread);	
功能概述	终止该线程运行，该线程将不再被 AKOS 调度，直到用户再次调用 ISubThread _ ReStart。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项	调用该接口后，线程状态改为 Terminate 状态。	
调用示例	ISubThread _ Terminate (p ISubThread);	

6.3.4 ISubThread _Exit

原　　型	T_ S32 ISubThread _ Exit (ISubThread *p ISubThread);	
功能概述	退出该线程，包括先终止该线程，然后从 AKOS 线程链表中删除该线程。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	ISubThread _ Exit (p ISubThread);	

6.3.5 ISubThread _ SetProperty

原　　型	T_ S32 ISubThread _ SetProperty (ISubThread *p ISubThread，T_U16 wPropID, T_VOID* pPropData);	
功能概述	设置线程属性值，如事件掩码等。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [in]	指向要设置的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例	ISubThread _ SetProperty (p ISubThread，wPropID，pPropData);	

6.3.6 ISubThread _ GetProperty

原　　型	T_ S32 ISubThread _ GetProperty (ISubThread *p ISubThread，T_U16 wPropID, T_VOID* pPropData);	
功能概述	获取线程属性值，如事件掩码等。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
	wPropID [in]	属性类别标识
	pPropData [out]	指向要获取的属性值
返回值说明	AK_SUCCESS	表示操作成功
	非 AK_SUCCESS	表示操作失败
注意事项		
调用示例		

6.3.7 SubThread _GetState

原　　型	T_U16 ISubThread _GetState (ISubThread *p ISubThread);	
功能概述	获取线程当前状态。	
参数说明	ISubThread [in]	指向该组件实例的接口指针
返回值说明	T_U16 类型值	表示线程当前状态
注意事项	线程状态有： THREAD_STATE_READY、 THREAD_STATE_WAITING、 THREAD_STATE_SUSPEND、 THREAD_STATE_TERMINATE 等	
调用示例	ISubThread _GetState (p ISubThread);	

6.3.8 CSubThread_New

原　　型	T_S32 CSubThread_New(ISubThread **ppi, T_SUBTHREAD_INITPARAM *pParam, T_BOOL bIsAutoRun);	
功能概述	创建子线程。	
参数说明	ppi [in/out]	返回线程实例的指针
	pParam [in]	子线程初始化参数
	bIsAutoRun [in]	是否要自动运行，如果为 AK_TRUE,则子线程创建好就投入运行。
返回值说明	T_U32 类型值	创建子线程返回值
注意事项		
调用示例	CSubThread_New(&pISubThread, &param, AK_TRUE);	

6.4 子线程实例

以 Image_Read 为例，子线程的使用步骤如下：

1） 添加子线程执行函数和异常销毁函数

```
Typedef struct tagImageRead
{
  T_hFile  hFile;
```

```
    T_TCHR szFileName[128+1];
    …  …
  }T_ImageRead;
  Void Image_Read_SubThread(T_ImageRead *pImageRead)
  {
      Image_Read（pImageRead-> szFileName,…）;
  }
  Void Image_Read_Abort(T_ImageRead *pImageRead)
  {
      //Do some free work…
  }
```

2） 创建一个子线程

```
T_S32 lRet                  = AK_SUCCESS;
ISubThread *pISubThread = AK_NULL;

T_SubThreadPatam param  =
{
    SUBTHREAD_MAINTHREAD_PRIORITY,  // Priority,
SUBTHREAD_MAINTHREAD_PRIORITY
//means the Priority is the same as its parent thread's Priority.
    2,                          // TimeSlice,
    5*1024,                     // StackSize
    AKAPP_CLSID_MMI,    //Parent thread class id.
    &imageread,                 //sub thread param, here should set the image path.
    Image_Read_SubThread,  //SubThread executive entity.
    Image_Read_Abort,       // SubThread free function.
};
lRet  =  CSubThread_New (&pISubThread  , &param, AK_TRUE);
if (AK_IS_SUCCESS(lRet) )
{
  //Ya, Success.
```

}

如果用户不需要在引用改子线程指针，则可以调用 AK_RELEASEIF(pISubThread)。这并不是真正释放了该线程，因为 CSubThread_New 会将该子线程挂到主线程的子线程列表里，当然子线程的引用计数也会加 1，所以这里调用 AK_RELEASEIF(pISubThread)只会让子线程引用计数减 1。

3） 挂起子线程（如果来电需要的话）

ISubThread_SusPend(pISubThread);

4） 恢复子线程

ISubThread_Resume(pISubThread);

5） 终止子线程（在当前任务没有执行完但需要执行一个新的任务时使用）

ISubThread_Terminate(pISubThread);

注：该子线程的 Abort 函数会被调用，如果有的话。

6） 重启子线程

ISubThread_ReStart(pISubThread, &imageread);

注：imageread 为 T_ImageRead 结构体变量。

7） 子线程同步

为了提高子线程的灵活性，没有将消息队列或事件集等封装到模版里，如果某个子线程确实需要，用户可以自己添加，添加步骤如下：（以队列为例）

a. 在子线程里创建一个消息队列,句柄为 hQueue。

b. 在子线程执行函数里循环从该队列获取事件。

如下所示：

```
Void Image_Read_SubThread(T_XxxParam *pParam)
{
    T_SYS_MAILBOX mailbox;
    While (1)
    {
```

```
            AK_Receive_From_Queue(pParam->hQueue,
                                  &mailbox,
                                  sizeof(T_SYS_MAILBOX)
                                  ,&ulActQuSize,
                                  AK_SUSPEND);
          Switch (mailbox.event)
           {
              Case  XXX_READ_IMAGE:
                    Image_Read(pParam->pParam);
                    Break;
              Case XXX_QUIT:
                  bIsQuit = AK_TRUE;
                    Break;
              Default:
                    Break;
           }
           If (bIsQuit)
            Break;
    }
}
```

c. 向子线程发送信息

```
AK_Send_To_Queue (hQueue,
                  &mailbox, sizeof(T_SYS_MAILBOX),
                  AK_SUSPEND);
```

注：如果一个应用退出，将销毁所有跟这个应用相关的子线程。

可能存在的问题：

将 Image_Read 放到子线程里，虽然能随时终止，但在下一次执行 Image_Read 时，由于前一次终止没有将解码器相关参数复位（因为 Image_Read 在子线程里仍是一块整体，只有通过 terminate 该任务来终止执行），所以有可能导致程序执行异常。