本章将继续探索驱动开发中的基础部分，定时器在内核中同样很常用，在内核中定时器可以使用两种，即IO定时器，以及DPC定时器，一般来说IO定时器是DDK中提供的一种，该定时器可以为间隔为N秒做定时，但如果要实现毫秒级别间隔，微秒级别间隔，就需要用到DPC定时器，如果是秒级定时其两者基本上无任何差异，本章将简单介绍`IO/DPC`这两种定时器的使用技巧。

首先来看IO定时器是如何使用的，IO定时器在使用上需要调用`IoInitializeTimer`函数对定时器进行初始化，但需要注意的是此函数每个设备对象只能调用一次，当初始化完成后用户可调用`IoStartTimer`让这个定时器运行，相反的调用`IoStopTimer`则用于关闭定时。

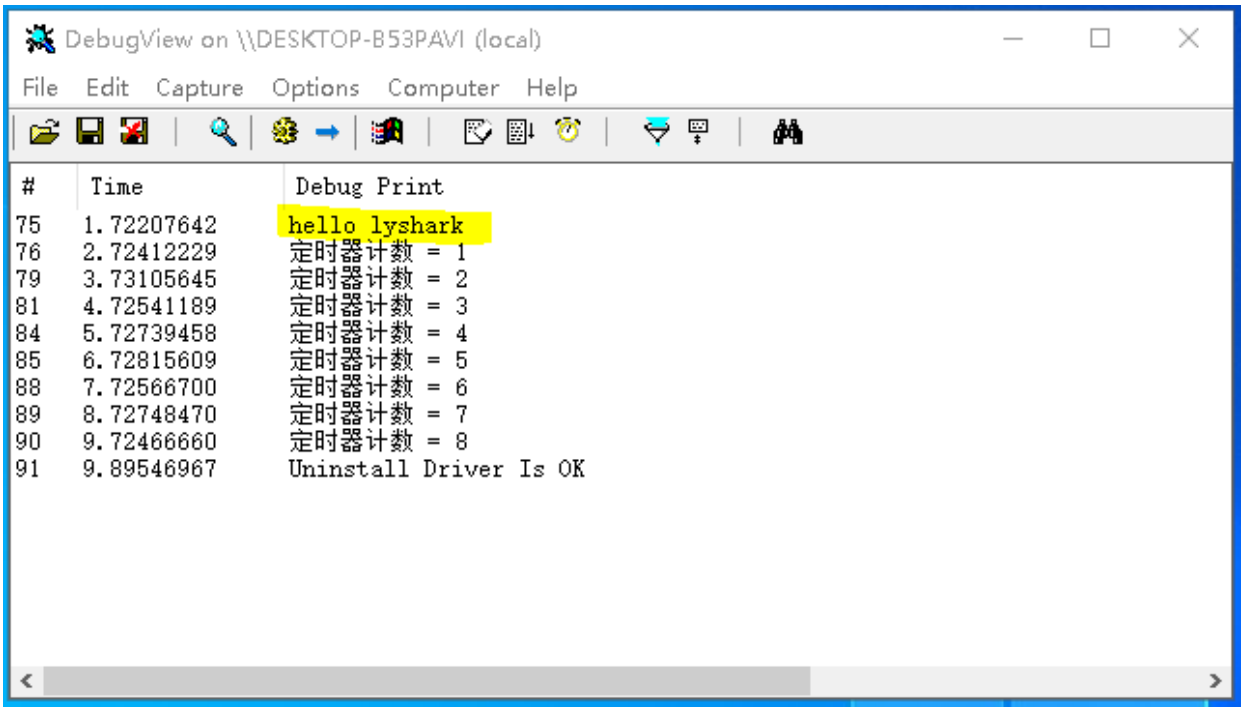
```
// 初始化定时器
NTSTATUS IoInitializeTimer(
  [in]           PDEVICE_OBJECT         DeviceObject,  // 设备对象
  [in]           PIO_TIMER_ROUTINE      TimerRoutine,  // 回调例程
  [in, optional] __drv_aliasesMem PVOID Context        // 回调例程参数
);
// 启动定时器
VOID IoStartTimer(
  [in] PDEVICE_OBJECT DeviceObject             // 设备对象
);
// 关闭定时器
VOID IoStopTimer(
  [in] PDEVICE_OBJECT DeviceObject             // 设备对象
);
```

这里我们最关心的其实是`IoInitializeTimer`函数中的第二个参数`TimerRoutine`该参数用于传递一个自定义回调函数地址，其次由于定时器需要依附于一个设备，所以我们还需要调用`IoCreateDevice`创建一个新设备来让定时器线程使用，实现定时器代码如下所示。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <wdm.h>
#include <ntstrsafe.h>

LONG count = 0;

// 自定义定时器函数
VOID MyTimerProcess( __in struct _DEVICE_OBJECT *DeviceObject, __in_opt PVOID
Context)
{
    InterlockedIncrement(&count);
    DbgPrint("定时器计数 = %d", count);
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    // 关闭定时器
```

```
    IoStopTimer(driver->DeviceObject);

    // 删除设备
    IoDeleteDevice(driver->DeviceObject);

    DbgPrint(("Uninstall Driver Is OK \n"));
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark \n");

    NTSTATUS status = STATUS_UNSUCCESSFUL;

    // 定义设备名以及定时器
    UNICODE_STRING dev_name = RTL_CONSTANT_STRING(L"");
    PDEVICE_OBJECT dev;
    status = IoCreateDevice(Driver, 0, &dev_name, FILE_DEVICE_UNKNOWN,
FILE_DEVICE_SECURE_OPEN, FALSE, &dev);
    if (!NT_SUCCESS(status))
    {
        return STATUS_UNSUCCESSFUL;
    }
    else
    {
        // 初始化定时器并开启
        IoInitializeTimer(dev, MyTimerProcess, NULL);
        IoStartTimer(dev);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

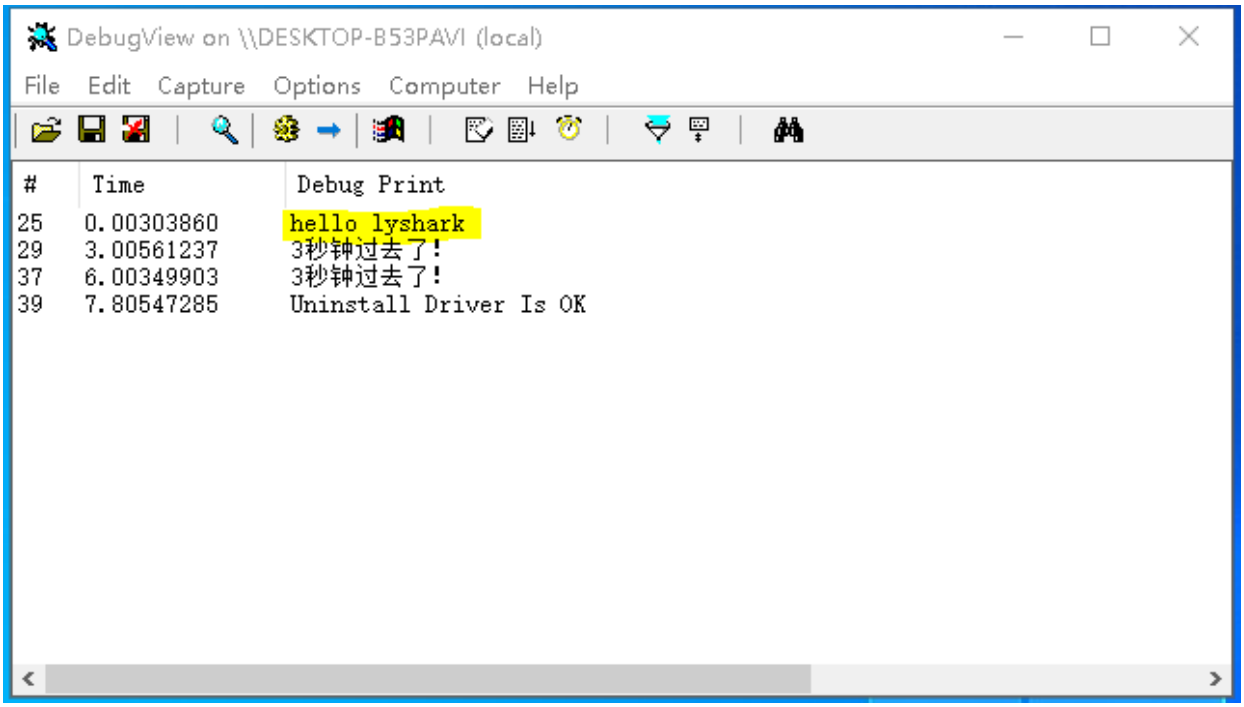
编译并运行这段代码，那么系统会每隔1秒执行一次`MyTimerProcess`这个自定义函数。



那么如何让其每隔三秒执行一次呢，其实很简单，通过`InterlockedDecrement`函数实现递减（每次调用递减1）当计数器变为0时`InterlockedCompareExchange`会让其继续变为3，以此循环即可完成三秒输出一次的效果。

```
LONG count = 3;

// 自定义定时器函数
VOID MyTimerProcess(__in struct _DEVICE_OBJECT *DeviceObject, __in_opt PVOID
Context)
{
    // 递减计数
    InterlockedDecrement(&count);

    // 当计数减到0之后继续变为3
    LONG preCount = InterlockedCompareExchange(&count, 3, 0);

    //每隔3秒计数器一个循环输出如下信息
    if (preCount == 0)
    {
        DbgPrint("[LyShark] 三秒过去了 \n");
    }
}
```

程序运行后，你会看到如下输出效果；



相比于IO定时器来说，DPC定时器则更加灵活，其可对任意间隔时间进行定时，DPC定时器内部使用定时器对象KTIMER，当对定时器设定一个时间间隔后，每隔这段时间操作系统会将一个DPC例程插入DPC队列。当操作系统读取DPC队列时，对应的DPC例程会被执行，此处所说的DPC例程同样表示回调函数。

DPC定时器中我们所需要使用的函数声明部分如下所示；

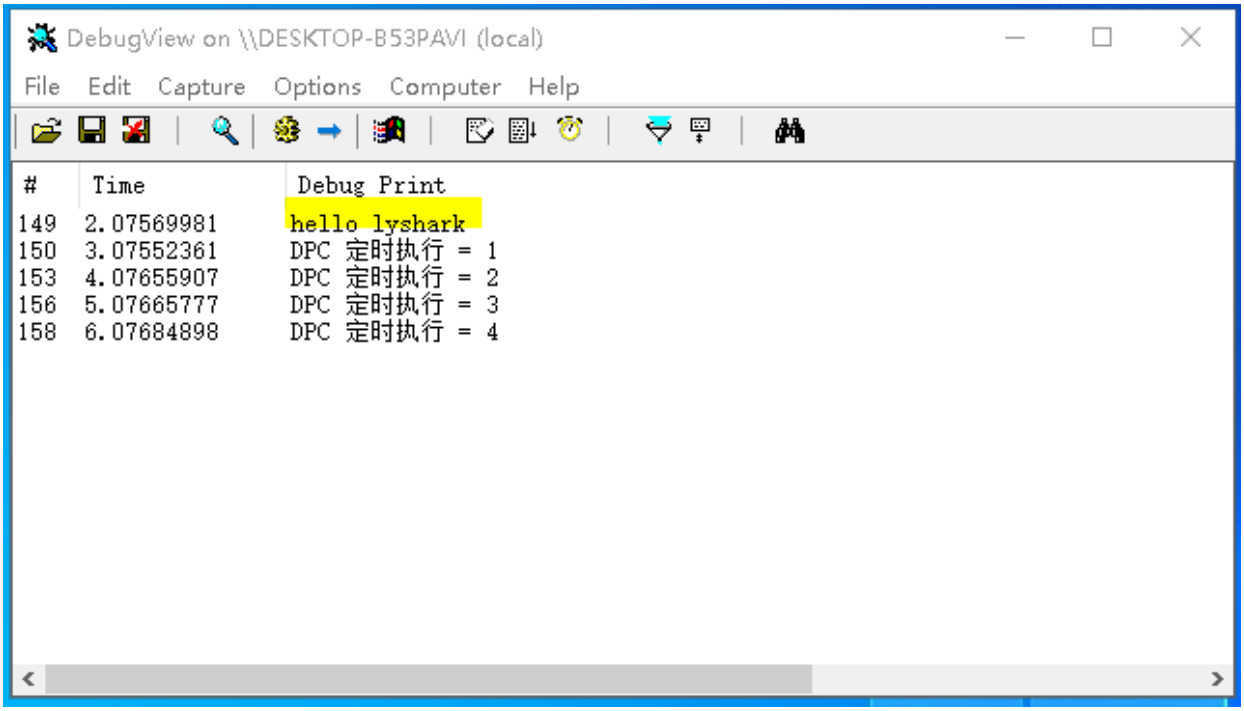
```
// 初始化定时器对象 PKTIMER 指向调用方为其提供存储的计时器对象的指针
void KeInitializeTimer(
  [out] PKTIMER Timer    // 定时器指针
);

// 初始化DPC对象
void KeInitializeDpc(
  [out]          __drv_aliasesMem PRKDPC Dpc,
  [in]           PKDEFERRED_ROUTINE      DeferredRoutine,
  [in, optional] __drv_aliasesMem PVOID  DeferredContext
);

// 设置定时器
BOOLEAN KeSetTimer(
  [in, out]      PKTIMER       Timer,     // 定时器对象的指针
  [in]           LARGE_INTEGER DueTime,   // 时间间隔
  [in, optional] PKDPC         Dpc        // DPC对象
);

// 取消定时器
BOOLEAN KeCancelTimer(
  [in, out] PKTIMER unnamedParam1         // 定时器指针
);
```

注意；在调用KeSetTimer后，只会触发一次DPC例程。如果想周期的触发DPC例程，需要在DPC例程被触发后，再次调用KeSetTimer函数，应用DPC定时代码如下所示。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <wdm.h>
#include <ntstrsafe.h>

LONG count = 0;
KTIMER g_ktimer;
KDPC g_kdpc;

// 自定义定时器函数
VOID MyTimerProcess(__in struct _KDPC *Dpc,__in_opt PVOID DeferredContext,__in_opt
PVOID SystemArgument1,__in_opt PVOID SystemArgument2)
{
    LARGE_INTEGER la_dutime = { 0 };
    la_dutime.QuadPart = 1000 * 1000 * -10;

    // 递增计数器
    InterlockedIncrement(&count);

    DbgPrint("DPC 定时执行 = %d", count);

    // 再次设置定时
    KeSetTimer(&g_ktimer, la_dutime, &g_kdpc);
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    // 取消计数器
    KeCancelTimer(&g_ktimer);

    DbgPrint(("Uninstall Driver Is OK \n"));
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark \n");

    LARGE_INTEGER la_dutime = { 0 };

    // 每隔1秒执行一次
    la_dutime.QuadPart = 1000 * 1000 * -10;

    // 1.初始化定时器对象
    KeInitializeTimer(&g_ktimer);

    // 2.初始化DPC定时器
    KeInitializeDpc(&g_kdpc, MyTimerProcess, NULL);
```

```
    // 3.设置定时器,开始计时
    KeSetTimer(&g_ktimer, la_dutime, &g_kdpc);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

编译并运行这段程序，会发现其运行后的定时效果与IO定时器并无太大区别，但是DPC可以控制更精细，通过`la_dutime.QuadPart = 1000 * 1000 * -10`毫秒级别都可被控制。

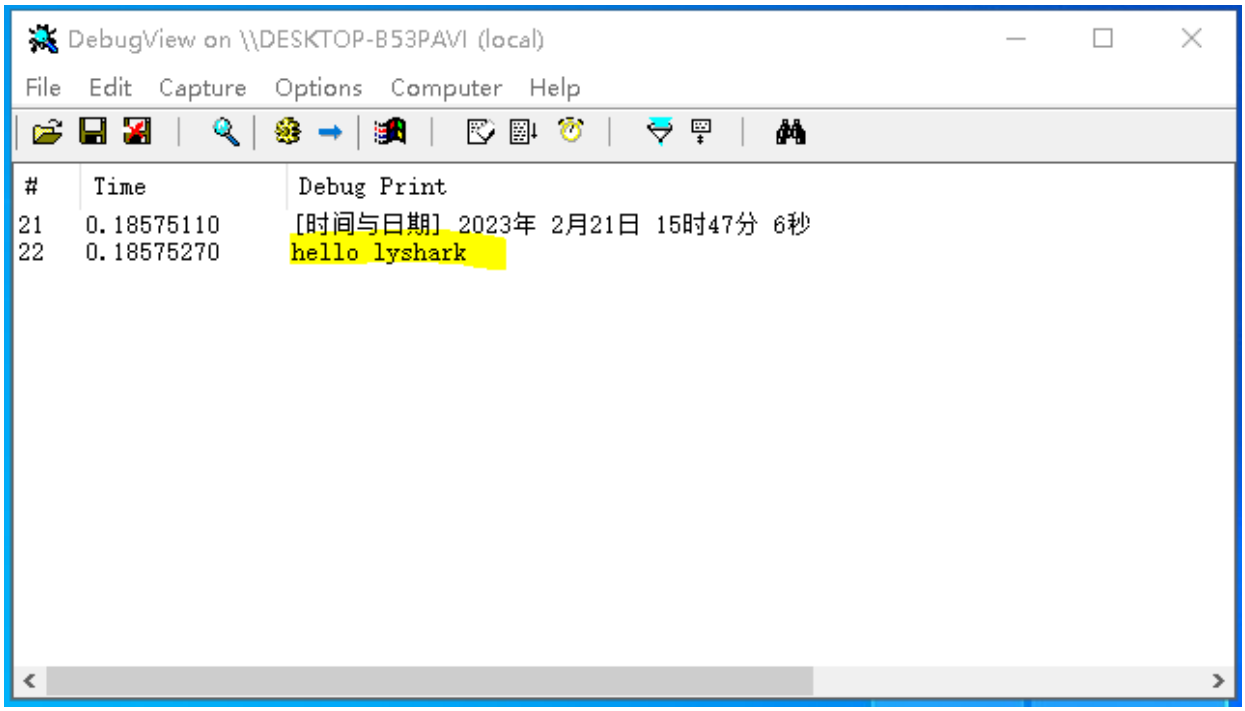


最后扩展一个知识点，如何得到系统的当前详细时间，获得系统时间。在内核里通过`KeQuerySystemTime`获取的系统时间是标准时间`（GMT+0）`，转换成本地时间还需使用`RtlTimeToTimeFields`函数将其转换为`TIME_FIELDS`结构体格式。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <wdm.h>
#include <ntstrsafe.h>

/*
    typedef struct TIME_FIELDS
    {
    CSHORT Year;
    CSHORT Month;
    CSHORT Day;
    CSHORT Hour;
    CSHORT Minute;
    CSHORT Second;
```

```
    CSHORT Milliseconds;
    CSHORT Weekday;
    } TIME_FIELDS;
*/

// 内核中获取时间
VOID MyGetCurrentTime()
{
    LARGE_INTEGER CurrentTime;
    LARGE_INTEGER LocalTime;
    TIME_FIELDS   TimeFiled;

    // 得到格林威治时间
    KeQuerySystemTime(&CurrentTime);

    // 转成本地时间
    ExSystemTimeToLocalTime(&CurrentTime, &LocalTime);

    // 转换为TIME_FIELDS格式
    RtlTimeToTimeFields(&LocalTime, &TimeFiled);

    DbgPrint("[时间与日期] %4d年%2d月%2d日 %2d时%2d分%2d秒",
        TimeFiled.Year, TimeFiled.Month, TimeFiled.Day,
        TimeFiled.Hour, TimeFiled.Minute, TimeFiled.Second);
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    MyGetCurrentTime();

    DbgPrint("hello lyshark \n");

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

运行后即可在内核中得到当前系统的具体时间;


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help
Time Debug Print
21 0.18575110 [时间与日期] 2023年 2月21日 15时47分 6秒
22 0.18575270 hello lyshark