如前所述，在前几章内容中笔者简单介绍了`内存读写`的基本实现方式，这其中包括了`CR3切换`读写，`MDL映射`读写，`内存拷贝`读写，本章将在如前所述的读写函数进一步封装，并以此来实现驱动读写内存浮点数的目的。内存`浮点数`的读写依赖于`读写内存字节`的实现，因为浮点数本质上也可以看作是一个字节集，对于`单精度浮点数`来说这个字节集列表是4字节，而对于`双精度浮点数`，此列表长度则为8字节。

如下代码片段摘取自本人的`LyMemory`驱动读写项目，函数`ReadProcessMemoryByte`用于读取内存特定字节类型的数据，函数`WriteProcessMemoryByte`则用于写入字节类型数据，完整代码如下所示；

这段代码中依然采用了`《驱动开发：内核MDL读写进程内存》`中所示的读写方法，通过MDL附加到进程并`RtlCopyMemory`拷贝数据，至于如何读写字节集只需要循环读写即可实现；

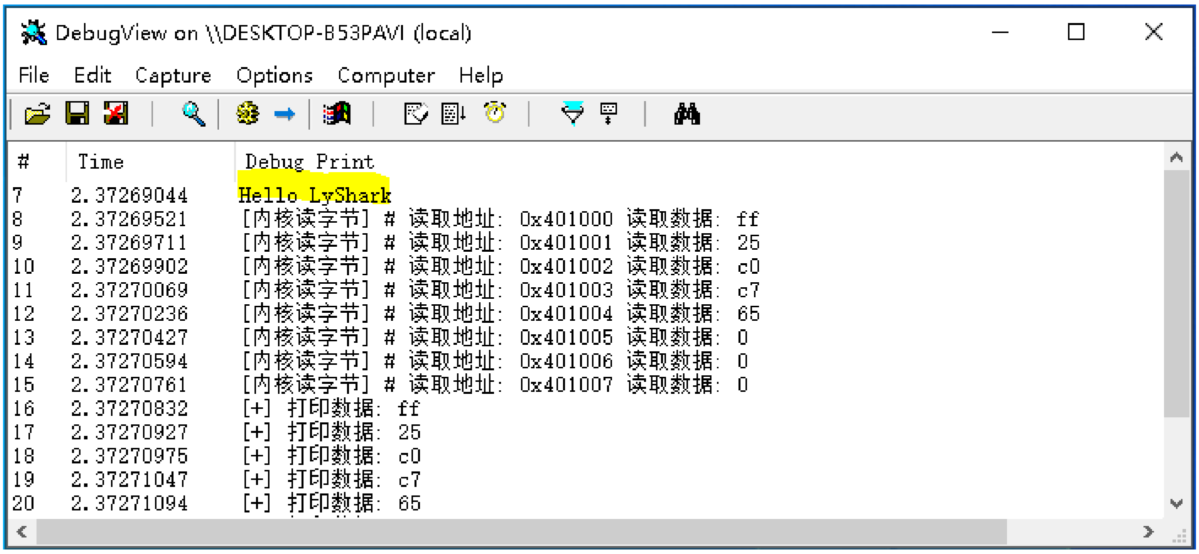
```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <windef.h>

// 读取内存字节
BYTE ReadProcessMemoryByte(HANDLE Pid, ULONG64 Address, DWORD Size)
{
    KAPC_STATE state = { 0 };
    BYTE OpCode;

    PEPROCESS Process;
    PsLookupProcessByProcessId((HANDLE)Pid, &Process);

    // 绑定进程对象,进入进程地址空间
    KeStackAttachProcess(Process, &state);

    __try
    {
        // ProbeForRead 检查内存地址是否有效, RtlCopyMemory 读取内存
        ProbeForRead((HANDLE)Address, Size, 1);
        RtlCopyMemory(&OpCode, (BYTE *)Address, Size);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        // 调用KeUnstackDetachProcess解除与进程的绑定,退出进程地址空间
        KeUnstackDetachProcess(&state);

        // 让内核对象引用数减1
        ObDereferenceObject(Process);
        // DbgPrint("读取进程 %d 的地址 %x 出错", ptr->Pid, ptr->Address);
        return FALSE;
    }

    // 解除绑定
    KeUnstackDetachProcess(&state);
    // 让内核对象引用数减1
    ObDereferenceObject(Process);
    DbgPrint("[内核读字节] # 读取地址: 0x%x 读取数据: %x \n", Address, OpCode);

    return OpCode;
```

```
}

// 写入内存字节
BOOLEAN WriteProcessMemoryByte(HANDLE Pid, ULONG64 Address, DWORD Size, BYTE
*OpCode)
{
    KAPC_STATE state = { 0 };

    PEPROCESS Process;
    PsLookupProcessByProcessId((HANDLE)Pid, &Process);

    // 绑定进程,进入进程的地址空间
    KeStackAttachProcess(Process, &state);

    // 创建MDL地址描述符
    PMDL mdl = IoAllocateMdl((HANDLE)Address, Size, 0, 0, NULL);
    if (mdl == NULL)
    {
        return FALSE;
    }

    //使MDL与驱动进行绑定
    MmBuildMdlForNonPagedPool(mdl);
    BYTE* ChangeData = NULL;

    __try
    {
        // 将MDL映射到我们驱动里的一个变量,对该变量读写就是对MDL对应的物理内存读写
        ChangeData = (BYTE *)MmMapLockedPages(mdl, KernelMode);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        // DbgPrint("映射内存失败");
        IoFreeMdl(mdl);

        // 解除映射
        KeUnstackDetachProcess(&state);
        // 让内核对象引用数减1
        ObDereferenceObject(Process);
        return FALSE;
    }

    // 写入数据到指定位置
    RtlCopyMemory(ChangeData, OpCode, Size);
    DbgPrint("[内核写字节] # 写入地址: 0x%x 写入数据: %x \n", Address, OpCode);

    // 让内核对象引用数减1
    ObDereferenceObject(Process);
    MmUnmapLockedPages(ChangeData, mdl);
    KeUnstackDetachProcess(&state);
    return TRUE;
}
```

实现读取内存字节集并将读入的数据放入到`LySharkReadByte`字节列表中，这段代码如下所示，通过调用`ReadProcessMemoryByte`都内存字节并每次`0x401000 + i`在基址上面增加变量i以此来实现字节集读取；

```
// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    // 读内存字节集
    BYTE LySharkReadByte[8] = { 0 };

    for (size_t i = 0; i < 8; i++)
    {
        LySharkReadByte[i] = ReadProcessMemoryByte(4884, 0x401000 + i, 1);
    }

    // 输出读取的内存字节
    for (size_t i = 0; i < 8; i++)
    {
        DbgPrint("[+] 打印数据: %x \n", LySharkReadByte[i]);
    }

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

运行如上代码片段，你会看到如下图所示的读取效果；

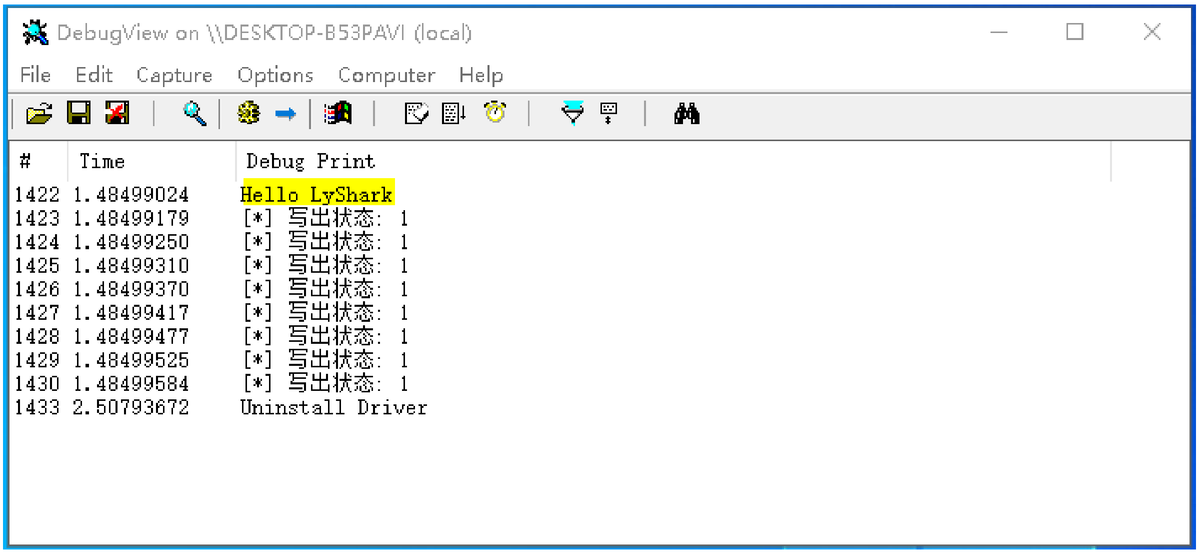


那么如何实现写内存字节集呢？其实写入内存字节集与读取基本类似，通过填充`LySharkWriteByte`字节集列表，并调用`WriteProcessMemoryByte`函数依次循环字节集列表即可实现写出字节集的目的；

```
// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    // 内存写字节集
```

```
    BYTE LySharkWriteByte[8] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
};

    for (size_t i = 0; i < 8; i++)
    {
        BOOLEAN ref = WriteProcessMemoryByte(4884, 0x401000 + i, 1,
LySharkWriteByte[i]);
        DbgPrint("[*] 写出状态: %d \n", ref);
    }

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

运行如上代码片段，即可将`LySharkWriteByte[8]`中的字节集写出到内存`0x401000 + i`的位置处，输出效果图如下所示；



接下来不如本章的重点内容，首先如何实现读内存`单精度与双精度`浮点数的目的，实现原理是通过读取BYTE类型的前4或者8字节的数据，并通过`*((FLOAT*)buffpyr)`将其转换为浮点数，通过此方法即可实现字节集到浮点数的转换，而决定是单精度还是双精度则只是一个字节集长度问题，这段读写代码实现原理如下所示；

```
// 读内存单精度浮点数
FLOAT ReadProcessFloat(DWORD Pid, ULONG64 Address)
{
    BYTE buff[4] = { 0 };
    BYTE* buffpyr = buff;

    for (DWORD x = 0; x < 4; x++)
    {
        BYTE item = ReadProcessMemoryByte(Pid, Address + x, 1);
        buff[x] = item;
    }

    return *((FLOAT*)buffpyr);
}

// 读内存双精度浮点数
DOUBLE ReadProcessMemoryDouble(DWORD Pid, ULONG64 Address)
{
```

```
    BYTE buff[8] = { 0 };
    BYTE* buffpyr = buff;

    for (DWORD x = 0; x < 8; x++)
    {
        BYTE item = ReadProcessMemoryByte(Pid, Address + x, 1);
        buff[x] = item;
    }

    return *((DOUBLE*)buffpyr);
}

// 驱动卸载例程
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    // 读取单精度
    FLOAT fl = ReadProcessFloat(4884, 0x401000);
    DbgPrint("[读取单精度] = %d \n", fl);

    // 读取双精度浮点数
    DOUBLE fl = ReadProcessMemoryDouble(4884, 0x401000);
    DbgPrint("[读取双精度] = %d \n", fl);

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

如上代码就是实现`浮点数`读写的关键所在，这段代码中的`浮点数`传值如果在内核中会提示`无法解析的外部符号 _fltused`此处只用于演示核心原理，如果想要实现不报错，该代码中的传值操作应在应用层进行，而传入参数也应改为字节类型即可。

同理，对于写内存浮点数而言依旧如此，只是在接收到用户层传递参数后应对其`dtoc`双精度浮点数转为CHAR或者`ftoc`单精度浮点数转为CHAR类型，再写出即可；

```
// 将DOUBLE适配为合适的Char类型
VOID dtoc(double dvalue, unsigned char* arr)
{
    unsigned char* pf;
    unsigned char* px;
    unsigned char i;

    // unsigned char型指针取得浮点数的首地址
    pf = (unsigned char*)&dvalue;

    // 字符数组arr准备存储浮点数的四个字节,px指针指向字节数组arr
    px = arr;

```

```
    for (i = 0; i < 8; i++)
    {
        // 使用unsigned char型指针从低地址一个字节一个字节取出
        *(px + i) = *(pf + i);
    }
}

// 将Float适配为合适的Char类型
VOID ftoc(float fvalue, unsigned char* arr)
{
    unsigned char* pf;
    unsigned char* px;
    unsigned char i;

    // unsigned char型指针取得浮点数的首地址
    pf = (unsigned char*)&fvalue;

    // 字符数组arr准备存储浮点数的四个字节,px指针指向字节数组arr
    px = arr;

    for (i = 0; i < 4; i++)
    {
        // 使用unsigned char型指针从低地址一个字节一个字节取出
        *(px + i) = *(pf + i);
    }
}

// 写内存单精度浮点数
BOOL WriteProcessMemoryFloat(DWORD Pid, ULONG64 Address, FLOAT write)
{
    BYTE buff[4] = { 0 };
    ftoc(write, buff);

    for (DWORD x = 0; x < 4; x++)
    {
        BYTE item = WriteProcessMemoryByte(Pid, Address + x, buff[x], 1);
        buff[x] = item;
    }

    return TRUE;
}

// 写内存双精度浮点数
BOOL WriteProcessMemoryDouble(DWORD Pid, ULONG64 Address, DOUBLE write)
{
    BYTE buff[8] = { 0 };
    dtoc(write, buff);

    for (DWORD x = 0; x < 8; x++)
    {
        BYTE item = WriteProcessMemoryByte(Pid, Address + x, buff[x], 1);
        buff[x] = item;
    }

    return TRUE;
}
```

```c
// 驱动卸载例程
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    // 写单精度
    FLOAT LySharkFloat1 = 12.5;
    INT fl = WriteProcessMemoryFloat(4884, 0x401000, LySharkFloat1);
    DbgPrint("[写单精度] = %d \n", fl);

    // 读取双精度浮点数
    DOUBLE LySharkFloat2 = 12.5;
    INT d1 = WriteProcessMemoryDouble(4884, 0x401000, LySharkFloat2);
    DbgPrint("[写双精度] = %d \n", d1);

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