

```

// BlockIO.sys main program: Driver for simple bulk read/write transfers, BETA Version 0.9.
//
// Derived from, and used with permission:
// Main program for loopback driver
// Copyright (C) 1999 by Walter Oney
// All rights reserved
//
//
// Copyright (C) 2001, Intel Corporation
// All rights reserved.
// Permission is hereby granted to merge this program code with other program
// material to create a derivative work. This derivative work may be distributed
// in compiled object form only. Any other publication of this program, in any form,
// without the explicit permission of the copyright holders is prohibited.
//
// Send questions and comments to John.Hyde@intel.com

#include "stdcls.h"
#include "driver.h"
#include <initguid.h>
#include "guid.h"
#include "ioctl.h"

#define MSGUSBSTRING(d,s,i) { \
    UNICODE_STRING sd; \
    if (i && NT_SUCCESS(GetStringDescriptor(d,i,&sd))) { DbgPrint(s, sd.Buffer); RtlFreeUnicodeString(&sd); }}

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo);
NTSTATUS GetStringDescriptor(PDEVICE_OBJECT fdo, UCHAR istring, PUNICODE_STRING s);
VOID DriverUnload(IN PDRIVER_OBJECT fdo);

BOOLEAN win98 = FALSE;

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

#pragma INITCODE

extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    KdPrint((DRIVERNAME "This is BETA software (V0.9)\nPlease register it at www.USB-By-Example.com\n"));

    // See if we're running under Win98 or Win2K
    win98 = IsWin98();
    KdPrint((DRIVERNAME "Running under Windows(R) ");
    if (win98) KdPrint(("98\n")); else KdPrint(("2000\n"));

    // Initialize function pointers
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;

    DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = DispatchReadWrite;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchReadWrite;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchControl;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = DispatchInternalControl;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;

    return STATUS_SUCCESS;
}

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

#pragma PAGEDCODE

VOID DriverUnload(PDRIVER_OBJECT DriverObject) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Unloading driver\n"));
}

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Adding Device\n"));
    NTSTATUS status;

    // Create a functional device object to represent the hardware we're managing.
    PDEVICE_OBJECT fdo;
    ULONG dxsize = (sizeof(DEVICE_EXTENSION) + 7) & ~7;
    ULONG xsize = dxsize + GetSizeofGenericExtension();

```

```

    status = IoCreateDevice(DriverObject, xsize, NULL, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE,
&fdo);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVERNAME "IoCreateDevice failed with status = %X\n", status));
        return status;
    }
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

// From this point forward, any error will have side effects that need to be cleaned up.
// Using a try-finally block allows us to modify the program easily without losing track of the side effects.
    __try {
        pdx->DeviceObject = fdo;
        pdx->Pdo = pdo;

// Declare the buffering method we'll use for read/write requests
        fdo->Flags |= DO_DIRECT_IO;

// Link our device object into the stack leading to the PDO
        pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
        if (!pdx->LowerDeviceObject) {
            KdPrint((DRIVERNAME "IoAttachDeviceToDeviceStack failed with status = \n"));
            status = STATUS_DEVICE_REMOVED;
            __leave;
        }

// Set power management flags in the device object
        fdo->Flags |= DO_POWER_PAGABLE;

// Initialize to use the GENERIC.SYS library
        pdx->pgx = (PGENERIC_EXTENSION) ((PUCHAR) pdx + dxsize);
        GENERIC_INIT_STRUCT gis = {sizeof(GENERIC_INIT_STRUCT)};
        gis.DeviceObject = fdo;
        gis.Pdo = pdo;
        gis.Ldo = pdx->LowerDeviceObject;
        gis.RemoveLock = &pdx->RemoveLock;
        gis.StartDevice = StartDevice;
        gis.StopDevice = StopDevice;
        gis.RemoveDevice = RemoveDevice;
        RtlInitUnicodeString(&gis.DebugName, LDRIVERNAME);
        gis.Flags = GENERIC_SURPRISE_REMOVAL_OK;

        status = InitializeGenericExtension(pdx->pgx, &gis);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVERNAME "InitializeGenericExtension failed with status = %X\n", status));
            __leave;
        }

// AddDevice was successful. Register the interface with a GUID
        status = GenericRegisterInterface(pdx->pgx, &BlockIO_GUID);
        if (!NT_SUCCESS(status)) KdPrint((DRIVERNAME "Could not register the GUID interface\n"));
// Clear the "initializing" flag so that we can get IRPs
        fdo->Flags &= ~DO_DEVICE_INITIALIZING;
    }
    __finally {
        // cleanup side effects
        if (!NT_SUCCESS(status)) {
            // need to cleanup
            if (pdx->devname.Buffer) RtlFreeUnicodeString(&pdx->devname);
            if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
            IoDeleteDevice(fdo);
        }
    }
    return status;
}

VOID RemoveDevice(IN PDEVICE_OBJECT fdo) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KdPrint((DRIVERNAME "Removing Device\n"));
    if (pdx->LowerDeviceObject) IoDetachDevice(pdx->LowerDeviceObject);
    IoDeleteDevice(fdo);
}

NTSTATUS StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "Starting Device\n"));
    NTSTATUS status;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    URB urb;

```

```

// Read our device descriptor. Check that we have the correct interface and get a handle for each pipe
UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST), USB_DEVICE_DESCRIPTOR_TYPE,
    0, 0, &pdx->dd, NULL, sizeof(pdx->dd), NULL);
status = SendAwaitUrb(fdo, &urb);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVERNAME "Error %X trying to read device descriptor\n", status));
    return status;
}

MSGUSBSTRING(fdo, DRIVERNAME " - Configuring device from '%ws'\n", pdx->dd.iManufacturer);
// Get Product Name. This is used to identify multiple BlockIO devices
UNICODE_STRING ProductName;
status = GetStringDescriptor(fdo, pdx->dd.iProduct, &ProductName);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVERNAME "Could not get required Product Name, Error = %x\n", status));
    return status;
}
pdx->ProductName.Buffer = ProductName.Buffer;
pdx->ProductName.MaximumLength = ProductName.MaximumLength;
RtlCopyUnicodeString(&pdx->ProductName, &ProductName);
RtlFreeUnicodeString(&ProductName);
KdPrint((DRIVERNAME "Product name is '%ws'\n", &pdx->ProductName.Buffer));

// Read the descriptor of the first configuration in two steps. First read the configuration descriptor header.
// The second step reads the configuration descriptor plus all embedded interface and endpoint descriptors.
USB_CONFIGURATION_DESCRIPTOR tcd;
UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
USB_CONFIGURATION_DESCRIPTOR_TYPE,
    0, 0, &tcd, NULL, sizeof(tcd), NULL);
status = SendAwaitUrb(fdo, &urb);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVERNAME "Error %X trying to read Configuration Descriptor header\n", status));
    return status;
}

ULONG size = tcd.wTotalLength;
PUSB_CONFIGURATION_DESCRIPTOR pcd = (PUSB_CONFIGURATION_DESCRIPTOR) ExAllocatePool(NonPagedPool, size);
if (!pcd) {
    KdPrint((DRIVERNAME "Unable to allocate %X bytes for Configuration Descriptor\n", size));
    return STATUS_INSUFFICIENT_RESOURCES;
}

__try {
    UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
USB_CONFIGURATION_DESCRIPTOR_TYPE,
        0, 0, pcd, NULL, size, NULL);
    status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVERNAME "Error %X trying to read Configuration Descriptor\n", status));
        return status;
    }

    MSGUSBSTRING(fdo, DRIVERNAME "Selecting configuration named '%ws'\n", pcd->iConfiguration);

// Locate the descriptor for the one and only interface we expect to find
PUSB_INTERFACE_DESCRIPTOR pid = USBD_ParseConfigurationDescriptorEx(pcd, pcd, -1, -1, -1, -1, -1);
ASSERT(pid);
MSGUSBSTRING(fdo, DRIVERNAME "Selecting interface named '%ws'\n", pid->iInterface);

// Create a URB to use in selecting a configuration.
USB_INTERFACE_LIST_ENTRY interfaces[2] = {
    {pid, NULL},
    {NULL, NULL}, // fence to terminate the array
};

PURB selurb = USBD_CreateConfigurationRequestEx(pcd, interfaces);
if (!selurb) {
    KdPrint((DRIVERNAME "Unable to create configuration request\n"));
    return STATUS_INSUFFICIENT_RESOURCES;
}

__try {

// Verify that the interface describes exactly the endpoints we expect
if (pid->bNumEndpoints != 2) {
    KdPrint((DRIVERNAME "%d is the wrong number of endpoints\n", pid->bNumEndpoints));
    return STATUS_DEVICE_CONFIGURATION_ERROR;
}
}
}

```

```

        PUSB_ENDPOINT_DESCRIPTOR ped = (PUSB_ENDPOINT_DESCRIPTOR) pid;
        ped = (PUSB_ENDPOINT_DESCRIPTOR) USBD_ParseDescriptors(pcd, tcd.wTotalLength, ped,
USB_ENDPOINT_DESCRIPTOR_TYPE);
        if (!ped || ped->bEndpointAddress != 0x82 || ped->bmAttributes != USB_ENDPOINT_TYPE_BULK ||
ped->wMaxPacketSize != 64) {
            KdPrint((DRIVERNAME "Endpoint has wrong attributes\n"));
            return STATUS_DEVICE_CONFIGURATION_ERROR;
        }
        ++ped;
        if (!ped || ped->bEndpointAddress != 0x2 || ped->bmAttributes != USB_ENDPOINT_TYPE_BULK ||
ped->wMaxPacketSize != 64) {
            KdPrint((DRIVERNAME "Endpoint has wrong attributes\n"));
            return STATUS_DEVICE_CONFIGURATION_ERROR;
        }
        ++ped;

        PUSBD_INTERFACE_INFORMATION pii = interfaces[0].Interface;
        ASSERT(pii->NumberOfPipes == pid->bNumEndpoints);

// Initialize the maximum transfer size for each of the endpoints
        pii->Pipes[0].MaximumTransferSize = TRANSFER_SIZE;
        pii->Pipes[1].MaximumTransferSize = TRANSFER_SIZE;
        pdx->maxtransfer = TRANSFER_SIZE; // save for use in handling reads & writes

// Submit the set-configuration request
        status = SendAwaitUrb(fdo, selurb);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVERNAME "Error %X trying to select configuration\n", status));
            return status;
        }

// Save the configuration and pipe handles
        pdx->hconfig = selurb->UrbSelectConfiguration.ConfigurationHandle;
        pdx->hinpipe = pii->Pipes[0].PipeHandle;
        pdx->houtpipe = pii->Pipes[1].PipeHandle;

// Transfer ownership of the configuration descriptor to the device extension
        pdx->pcd = pcd;
        pcd = NULL;
    }
    __finally {
        ExFreePool(selurb);
    }
    __finally {
        if (pcd) ExFreePool(pcd);
    }

    return STATUS_SUCCESS;
}

VOID StopDevice(IN PDEVICE_OBJECT fdo, BOOLEAN oktouch /* = FALSE */) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

// If it's okay to touch our hardware (ie we're processing an IRP_MN_STOP_DEVICE), deconfigure the device.
    if (oktouch) {
        URB urb;
        UsbBuildSelectConfigurationRequest(&urb, sizeof(_URB_SELECT_CONFIGURATION), NULL);
        NTSTATUS status = SendAwaitUrb(fdo, &urb);
        if (!NT_SUCCESS(status)) KdPrint((DRIVERNAME "Deconfigure device returned status %x\n", status));
    }

    if (pdx->pcd) ExFreePool(pdx->pcd);
    pdx->pcd = NULL;
}

NTSTATUS DispatchCreate(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "In DispatchCreate\n"));
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    InterlockedIncrement(&pdx->handles);
    return CompleteRequest(Irp, STATUS_SUCCESS, 0);
}

NTSTATUS DispatchClose(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    KdPrint((DRIVERNAME "In DispatchClose\n"));
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

```

```

PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
InterlockedDecrement(&pdx->handles);
return CompleteRequest(Irp, STATUS_SUCCESS, 0);
}

// Declare some support routines
NTSTATUS SendAwaitUrb(PDEVICE_OBJECT fdo, PURB urb) {
    PAGED_CODE();
    ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

    KEVENT event;
    KeInitializeEvent(&event, NotificationEvent, FALSE);

    IO_STATUS_BLOCK iostatus;
    PIRP Irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
        pdx->LowerDeviceObject, NULL, 0, NULL, 0, TRUE, &event, &iostatus);

    if (!Irp) {
        KdPrint((DRIVERNAME "Unable to allocate IRP for sending URB\n"));
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
    stack->Parameters.Others.Argument1 = (PVOID) urb;
    NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (status == STATUS_PENDING) {
        KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
        status = iostatus.Status;
    }
    return status;
}

NTSTATUS GetStringDescriptor(PDEVICE_OBJECT fdo, UCHAR istring, PUNICODE_STRING s) {
    // NOTE: a side-effect of this routine is the allocation of some memory from the PagedPool
    // The caller MUST free this memory
    PAGED_CODE();
    NTSTATUS status;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    URB urb;
    UCHAR data[256]; // maximum-length buffer

    // If this is the first time here, read string descriptor zero and arbitrarily select
    // the first language identifier as the one to use in subsequent get-descriptor calls.
    if (!pdx->langid) { // determine default language id
        UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
        USB_STRING_DESCRIPTOR_TYPE,
            0, 0, data, NULL, sizeof(data), NULL);
        status = SendAwaitUrb(fdo, &urb);
        if (!NT_SUCCESS(status)) return status;
        pdx->langid = *(LANGID*)(data + 2);
    }

    // Fetch the designated string descriptor.
    UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST), USB_STRING_DESCRIPTOR_TYPE,
        istring, pdx->langid, data, NULL, sizeof(data), NULL);
    status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status)) return status;

    ULONG nchars = (data[0] - 2) / 2;
    PWSTR p = (PWSTR) ExAllocatePool(PagedPool, data[0]);
    if (!p) return STATUS_INSUFFICIENT_RESOURCES;

    memcpy(p, data + 2, nchars*2);
    p[nchars] = 0;

    s->Length = (USHORT) (2 * nchars);
    s->MaximumLength = (USHORT) ((2 * nchars) + 2);
    s->Buffer = p;

    return STATUS_SUCCESS;
}

VOID ResetDevice(PDEVICE_OBJECT fdo) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KEVENT event;
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    IO_STATUS_BLOCK iostatus;

```

```

PIRP Irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_RESET_PORT,
    pdx->LowerDeviceObject, NULL, 0, NULL, 0, TRUE, &event, &iostatus);
if (!Irp) return;

NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
if (status == STATUS_PENDING) {
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
    status = iostatus.Status;
}

if (!NT_SUCCESS(status)) KdPrint((DRIVERNAME "ResetDevice returned status %x\n", status));
}

NTSTATUS ResetPipe(PDEVICE_OBJECT fdo, USB_PIPE_HANDLE hpipe) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    URB urb;
    urb.UrbHeader.Length = (USHORT) sizeof(_URB_PIPE_REQUEST);
    urb.UrbHeader.Function = URB_FUNCTION_RESET_PIPE;
    urb.UrbPipeRequest.PipeHandle = hpipe;

    NTSTATUS status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status)) KdPrint((DRIVERNAME "ResetPipe returned status = %x\n", status));
    return status;
}

VOID AbortPipe(PDEVICE_OBJECT fdo, USB_PIPE_HANDLE hpipe) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    URB urb;
    urb.UrbHeader.Length = (USHORT) sizeof(_URB_PIPE_REQUEST);
    urb.UrbHeader.Function = URB_FUNCTION_ABORT_PIPE;
    urb.UrbPipeRequest.PipeHandle = hpipe;

    NTSTATUS status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status)) KdPrint((DRIVERNAME "AbortPipe returned status = %x\n", status));
}

// Use GENERIC.SYS to handle PnP and Power Events
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return GenericDispatchPnp(pdx->pgx, Irp);
}

NTSTATUS DispatchPower(PDEVICE_OBJECT fdo, PIRP Irp) {
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return GenericDispatchPower(pdx->pgx, Irp);
}

////////////////////////////////////
#pragma LOCKEDCODE

NTSTATUS DispatchControl(PDEVICE_OBJECT fdo, PIRP Irp) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    // Check for my "WhoAreYou" IOCTL
    ULONG IoControlCode = stack->Parameters.DeviceIoControl.IoControlCode;
    KdPrint((DRIVERNAME "IOCTL %4.4x detected\n", IoControlCode));
    if (IoControlCode == IOCTL_GET_PRODUCT_NAME) {
        ULONG BufferLength = stack->Parameters.DeviceIoControl.OutputBufferLength;
        ULONG ActualLength = 0; // Default
        if (BufferLength < pdx->ProductName.Length) status = STATUS_INVALID_BUFFER_SIZE;
        else {
            ActualLength = pdx->ProductName.Length;
            memcpy(Irp->AssociatedIrp.SystemBuffer, pdx->ProductName.Buffer, ActualLength);
            status = STATUS_SUCCESS;
        }
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
        return CompleteRequest(Irp, status, ActualLength);
    }
    else { // Not mine, pass this down the stack
        IoSkipCurrentIrpStackLocation(Irp);
        status = IoCallDriver(pdx->LowerDeviceObject, Irp);
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    }
}

```

```

        return status;
    }
}

NTSTATUS DispatchInternalControl(PDEVICE_OBJECT fdo, PIRP Irp) {
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}

NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR info) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

extern "C" void __declspec(naked) __cdecl _chkesp()
{
    _asm je okay
    ASSERT(!DRIVERNAME " - Stack pointer mismatch!");
okay:
    _asm ret
}

```