WVIDEO Overlay Manager Interface

VERSION 1.0

January 26, 1994

Table of Contents

Overlay manager interface

For WVIDEO to be able to debug overlays, it must be able to make requests of the overlay manager for certain operations. The overlay manager must also be able to inform WVIDEO when a new overlay section is loaded.

When WVIDEO loads a DOS program, it looks at the initial CS:IP value for the following structure:

```
struct ovl_header {
    unsigned_8  short_jmp_opcode;       /* == 0xeb */
    signed_8    short_jmp_displacment;
    unsigned_16 signature;              /* == 0x2112 */
    void        (far *hook)();
    unsigned_16 handler_offset;
};
```

WVIDEO checks to make sure that the first instruction is a short jump (opcode 0xeb) and that the word following that instruction contains the value 0x2112. If this occurs, WVIDEO assumes that it is debugging an overlaid application.

WVIDEO then fills in the `hook` field with the far address of a routine that is invoked with a far call whenever a change in the overlay state occurs. The initial CS value and the contents of the `handler_offset` field gives the far address of the overlay manager routine responsible for handling debugger requests.

1 The Hook Routine

After the routine addresses have been exchanged, WVIDEO starts the program executing, to allow the overlay manager to initialize. After the manager has finished its initialization, it performs a far call to the debugger hook routine, with the return address on the stack being the "real" starting address of the program being debugged. All register contents (including flags) should be preserved by the hook routine.

After initialization, the debugger hook routine is invoked with a far call every time a new overlay section is loaded into memory. In this case the AX register contains the section number that was just loaded. The DL register contains a zero or non-zero value if the overlay load was caused by a call or return, respectively. The CX:BX registers form a far pointer to the last byte of the call instruction that caused the overlay load (in the case of a overlay load being caused by a return instruction (DL is non-zero) the far pointer is to the last byte of the call instruction that the return is returning from.)

NOTE: More sections than just the one identified by the section number in AX may be loaded by the overlay manager before the hook routine is called. The current overlay manager also loads all of the ancestors of a section (See the WLINK documentation in the Users' Guide for a description of

what an ancestor is). To find out what sections are really in memory the debugger should invoke the handler routine with a GET_OVERLAY_STATE request.

2 The Handler Routine

The handler routine is responsible for processing requests from the debugger pertaining to overlays. It is invoked by the debugger by performing a far call with a request number in the AX register. The AX register is used to return the result or return status of the request. The CX and BX registers are used for some requests to pass a far pointer to memory.

There are two structures that the handler routines deals with. The first is called an overlay state. An overlay state consists of a block of memory containing all the information necessary for the overlay manager to restore the overlays to their current condition at some later point in time. The first portion of this block is a bit vector, with each bit representing an overlay section. If the bit is a one, then the overlay section is currently in memory. If the bit is a zero then the overlay section is not in memory. To convert from a section number to a bit position use the following formulas:

```
byte_offset = (section_number - 1) / 8;
bit_number  = (section_number - 1) % 8;
```

Following the bit vector is information that the manager uses to restore the overlay stack.

The second structure used is an overlay address. This consists of a far pointer followed by a 16-bit section number.

The following requests are recognized by the debug handler routine.

2.1 GET_STATE_SIZE

```
Inputs:                         Outputs:
AX = request number (0)         AX = size of overlay state
```

This request returns the number of bytes required for an overlay state.

2.2 GET_OVERLAY_STATE

```
Inputs:                         Outputs:
AX = request number (1)         AX = 1
CX:BX = far pointer to memory
        to store overlay state
```

This request copies the overlay state into the memory pointed at by the CX:BX registers. A one is always returned in AX.

2.3 SET_OVERLAY_STATE

```
Inputs:                          Outputs:
AX = request number (2)          AX = 1
CX:BX = far pointer to memory
        to load overlay state
```

This request takes a previously obtained overlay state and causes the overlay manager to return itself to that overlay configuration. A one is always returned in AX. The overlay manager will not explicitly unload a section that is not in memory according to the given overlay state, so a GET_OVERLAY_STATE request following a SET_OVERLAY_STATE may not return the same bit vector portion. This request may also be used by the debugger to explicitly load a section, so the assembly code may be examined, perhaps. To do this, zero out a block of memory the size of an overlay state, and then turn on the appropriate section number in the bit vector, then make a SET_OVERLAY_STATE request. Remember that not only that section will be loaded, but all of its ancestor sections as well.

2.4 TRANSLATE_VECTOR_ADDR

```
Inputs:                          Outputs:
AX = request number (3)          AX = 1 if addr was translated,
CX:BX = far pointer to                0 otherwise
        overlay address
```

This request checks to see if the far pointer portion of the overlay address pointed at by CX:BX is actually an overlay vector. If the address is a vector then the vector address is replaced by the true address of the routine that the vector is for, and the section number portion is filled in with the section number the of routine. A one is returned in AX in this case. If the address is not an overlay vector, then the overlay address is untouched and an zero is returned in AX.

2.5 TRANSLATE_RETURN_ADDR

```
Inputs:                          Outputs:
AX = request number (4)          AX = 1 if addr was translated,
CX:BX = far pointer to                0 otherwise
        overlay address
```

In order to handle parallel overlay calls, the overlay manager replaces the true return address on the stack with that of some special code (the parallel return code). It then takes the original return address and section number an places them on the overlay stack. When a routine returns to the overlay manager, it pops the top entry of the overlay stack, makes sure that the original overlay section is loaded, and returns to the original return address.

This function performs much the same function as TRANSLATE_VECTOR_ADDR, except that rather than checking for a vector address, it checks to see if the address is that of the overlay manager parallel return code. If it is then the section number in the overlay address is used as the number of entries down in the overlay stack that the real return address and section number is to be found (zero is the top entry of the

overlay stack). The true return address and section number then replaces the contents of the overlay address and a one is returned in AX. If the address is not the parallel return code, then the overlay address is left untouched and a zero is returned in AX.

2.6 GET_OVL_TBL_ADDR

```
Inputs:                         Outputs:
AX = request number (5)         AX = 0
CX:BX = far pointer to variable
        of type far pointer to
        be filled in with
        overlay table address
```

This request fills in the far pointer pointed at by CX:BX with the address of the overlay table so that a profiler can find out where sections are located in the executable, or overlay files. The sampler program, when it detects that it is sampling a overlaid application, can perform this function and write the result into the sample file. Since the overlay table is always in the root, the profiler can then find the overlay table and from that, find the other sections. It should be noted that the format of the overlay table may change, so this call should be avoided if at all possible.

2.7 GET_MOVED_SECTION

```
Inputs:                         Outputs:
AX = request number (6)         AX = 1 if the section exists
CX:BX = far pointer to               0 otherwise
        overlay address
```

With the dynamic overlay manager, sections may be loaded, or moved, to positions other than where the linker originally placed them. The debugger must be informed of the new positions so that it can update the locations of its symbolic information. The GET_MOVED_SECTION request is responsible for informing the debugger what sections have moved and their new locations. The debugger will call this request after the hook routine has been called, or the debugger has invoked the SET_OVERLAY_STATE request. The request returns the first section whose id larger than the section number that is in the overlay address being passed in. The overlay manager will fill in the overlay address with the section number that has moved and its new segment address. The offset portion of the overlay address is unused. The request will return a one in AX. If there are no sections numbers larger than the one being passed in that have moved, a zero is returned.

Here is some example debugger code:

```
void CheckMovedSections()
{
    overlay_address     addr;

    addr.sect_id = 0;
    while( OvlHandler( GET_MOVED_SECTION, &addr ) ) {
        HandleMovedSection( addr.sect_id, addr.segment );
    }
}
```

2.8 GET_SECTION_DATA

```
Inputs:                          Outputs:
AX = request number (7)          AX = 1 if the section exists
CX:BX = far pointer to                0 otherwise
        overlay address
```

This request returns information on the current location of a section while it is in memory (or where it would be if it was loaded). The section number portion of the overlay address is filled in with the section id that information is being requested about before the request is made. The overlay manager returns zero in AX if the section does not exist. Otherwise it returns one and fills in the overlay address with the location that the section is in memory, or where it would currently go if it was loaded at that time. It also fills in the section number portion of the address with the size of the section in paragraphs.

3 Overlay Table Structure

The pointer returned by the GET_OVL_TBL_ADDR request has the following format:

```
typedef struct ovl_table {
    unsigned_8      major;
    unsigned_8      minor;
    void            far *start;
    unsigned_16     delta;
    unsigned_16     ovl_size;
    ovltab_entry    entries[ 1 ];
} ovl_table;
```

The fields `major` and `minor` field contain version numbers for the overlay table structure. If an upwardly compatible change in the structures is made, the minor number will be incremented. If a non-upwardly compatible change to the structures is made, the major field will be incremented. The current major version is three, the current minor version is zero. The `start` field contains a 32-bit far pointer to the "actual" starting address of the program. The overlay manager jumps to this address after it has finished initializing. (If a debugger/sampler is present then the overlay manager calls into the hook routine with this address on the return stack.) The `delta` field contains the value to be added to each of the segment relocations when a section is loaded into memory (it contains the segment value for the first segment in the program.) The `ovl_size` field contains the size of the overlay area. This is only used in the dynamic overlay manager. The final field, `entries`, is a variable sized array containing one entry for each overlay section in the program (e.g. the tenth element in the array describes overlay section 10.) Each entry has the following form:

```
typedef struct ovltab_entry {
    unsigned_16   flags_anc;
    unsigned_16   relocs;
    unsigned_16   start_para;
    unsigned_16   code_handle;
    unsigned_16   num_paras;
    unsigned_16   fname;
    unsigned_32   disk_addr;
} ovltab_entry;
```

The top bit of the `flag_anc` field contains an indicator, while the program is running, of whether the overlay section is in memory (value one) or must be loaded from disk (value zero). The next highest bit is filled in by the linker and informs the overlay manager that the section must be loaded during the overlay manager initialization. The remaining bits contain the overlay number for the ancestor of this section (zero if there is none). The `relocs` field say how many segment relocation items there are for this section, while the `start_para` field gives the location in memory (relative to the start of the program) that the section should be placed when loaded. The `num_paras` field contains the size of the section in paragraphs, and the `code_handle` field is used for various purposes inside the dynamic overlay loader. The `fname` field has the offset of the address of a zero terminated string for the name of the file containing the overlay section data and relocations (The segment value is the same as the overlay table). If the top bit of the offset is on, then the file is the original EXE file rather than a separate overlay file, and the overlay manager should use the program file name obtained from DOS (if the version is 3.0 or greater). The `disk_addr` field gives the starting offset the overlay data in the overlay file. The segment relocation items immediately follow the data.

The end of the `entries` array is indicated when an element's `flags_anc` field contains the value 0xffff. The remaining fields in that element contain garbage values.