Datalight®

RXE Theory of Operation

Contents

Document History

Revision	*Action*	*Author*	*Date*
0.90	First draft.	Dennis Edwards	10/24/2000
0.99	Final draft.	Dennis Edwards	10/28/2000
1.00	Formatting changes and final review	Brandon Thomas Keith Garvin	10/28/2000
1.01	Added Verification section and completed content.	Keith Garvin	12/01/2000

Introduction

For a variety of reasons, including but not limited to the 64K maximum segment size limit imposed by the Intel architecture, a DOS .EXE program will typically contain multiple program segments. The address of any program element (function entry point, variable, etc.) is determined by combining the physical location of the start of a segment (a 16-bit value expressed in units of 16 byte "paragraphs") with the 16-bit byte offset of the data element within that containing segment. The maximum size of a segment is limited by the 16 bits of the offset value and is 2^16 bytes or 64KB. The maximum real mode address space is also limited by the 16 bits of the segment value which is (64K paragraphs*16 bytes/paragraph) = 1MB.

Because of the segmented addressing scheme, a program designer must employ one of two strategies in order for a program to be relocatable at run-time. The choice of these strategies by the program designer determines whether the linker can generate a .COM file or if it must generate a .EXE file.

In a .COM (.BIN, etc.) file, all program elements are contained within a single segment. The value of this single segment can be determined at run-time by inspecting the value of the CS register which is establish by the DOS program loader when the program is passed control by the operating system. The initial entry point for program execution is firmly established as offset 100h within the single .COM file segment. This address represents the first possible program element location following the Program Segment Prefix (PSP). A DOS PSP is often generically referred to as a process control block in discussions of operating system theory.

The DOS PSP is a 256-byte data structure created by DOS as part of the load process. DOS updates the PSP during program execution. A DOS PSP contains such information as the number and value of file handles opened by the program, the program's command line arguments (if any), the segment value of the program's environment strings and so on. Because the PSP is located at offset 0 within the single .COM file segment the maximum effective size of a .COM file image is 2^16-256=65280 bytes.

In order to overcome the size limit of .COM files or to otherwise take advantage of the segmented Intel architecture (yes, there ARE advantages to it) the program designer may chose to employ the alternate .EXE file strategy. A DOS .EXE file may contain multiple code (executable) and/or data segments and the program image may be as large as the real mode environment and available memory allow. A .EXE file, like a .COM file, is fully run-time relocatable by the DOS operating system.

When DOS runs a normal .EXE file from a disk file, the initial load image, minus overlays, etc. but including both code and data is loaded into RAM from disk. The location and length of the initial load image is stored in the file's .EXE header that is prepended to the program load image by the linker. The .EXE file header also contains a list of “fix-ups” (a.k.a. program relocatable entries.)

Fix-Up Descriptions

A fix-up is a datum that identifies the location, relative to the start of the load image, of a segment reference: The value in the .EXE load image found at the fix-up location is the paragraph address relative to the start of the load image where the referenced program segment begins.

In order for standard multi-segment .EXE files to be fully relocatable at run-time, DOS must "fix up" the paragraph addresses in the .EXE image with the actual segment addresses of the program when it loads the program image from disk. The actual segment address of the start of a segment is the paragraph address within the image plus the segment value at which the program is loaded. Within a .EXE file, offsets are always taken relative to the start of the element's containing segment rather than the program's PSP segment. The segment-relative addressing scheme used by .EXE files is convenient for building overlaid code modules as well as iterated control constructs such as static object constructor and destructor tables.

In many embedded computer systems, mechanical disk drives are emulated by a combination data structures and code that are contained in ROM. DOS ROM disks exist within the executable first megabyte of address space of Intel processors operating in Real Mode. Because programs routinely need to update data as part of their normal execution, an ordinary .COM or .EXE file must first be copied to RAM by DOS before the program can executed. When execution space is at a premium, these duplicated ROM and RAM images are wasteful of system resources. If a scheme could be developed to execute at least part of the program directly from ROM, without copying that portion of the program image to RAM, then embedded system manufacturers could realize significant space savings. It is precisely this problem of wasted system resources that the Datalight ROMable EXecutable (RXE) conversion program solves.

In order to achieve the greatest user benefit, the RXE conversion program relies on the segment ordering conventions of DOS programs that demand that the .EXE image be divided into Code and Data segment classes with all data segments grouped together at the end of the image file. Any Code segment has executable and readable but not writable attributes and so may contain both executable code and constant data. Any program element that must be modified during program execution must be contained within the collection of data segments located at the end of the .EXE image file. An obvious assumption of the RXE conversion program is a complete absence of self-modifying code. This ban on self-modifying code also disallows run-time variables within the Code segment. These restrictions on Code segment attributes are typical of modern operating system protection schemes even though commercial DOS compiler start-up code and floating point emulation software often violate such commonly accepted tenants.

Removing Fix-Ups

Code that runs directly from ROM is sometimes referred to as "eXecute In Place" (XIP) code. In order for the RXE conversion program to obtain XIP capability it must consider three classes of fix-ups. The first class of fix-ups are those which refer to Code segments. Since all Code segments will remain in a fixed ROM location, any Code relative fix-up identified in the .EXE header can be replaced with a constant segment address value and then removed from the fix-up list in the .EXE file header of the resultant .RXE program file. All Code segment references, whether contained in a Code or a Data segment, are dealt with in this way. The constant values that are used to replace the Code relative fix-ups are based on information provided by the user when the RXE conversion program is run. The user must specify both the segment address value where the XIP image (including the RXE header) will reside in ROM as well as the name (as specified in the program's .MAP file) of the first Data segment in the .EXE image file that has a writable segment attribute.

The result of the RXE conversion process is a partially relocatable .RXE image file. The program entry point of a .RXE file, specified in the .EXE file header, will point to executable code in ROM. The .EXE file header of the .RXE image file is also modified so that only the .RXE Data segments (and the RXE_BASE segment which proceeds them) are loaded into RAM and fixed up by DOS in the normal manner. The fix-ups that are performed by DOS at load time are the second class of fix-ups that must be considered by the RXE conversion program. Each of these fix-ups that resides in a Data segment and refers to a data element contained within the relocatable Data segments of the .RXE program. The length of the .EXE header is adjusted to include the Code segments of the program so that only the Data and RXE_BASE segments are loaded into RAM and fixed up by DOS. The amount of RAM space saved by this XIP technique depends on the amount of executable code and constant data that are stored in the Code segments of the .EXE file since only those segments identified by the user as having a writable attribute are copied to RAM by the DOS program loader.

The third, final, and most troublesome class of fix-ups are those contained within Code segments and that refer to program elements contained within a Data segment. Since these Data segment references reside in Code segments that are physically read-only, DOS cannot fix them up at load time. Also, since various run-time factors (number of and nature of DOS device drivers, internal DOS data structure allocations, space reserved for user environment strings, etc.) each influence the location at which the relocatable Data segments are loaded into memory by DOS, these Data relative fix-ups cannot be resolved solely by the RXE conversion program at the time it is run, run-time steps must also be taken to adjust these addresses.

Handling Runtime Fix-Ups

The solution employed by the RXE conversion program for this third class of fix-ups is to replace the machine code instructions that access the data with a three-byte instruction sequence. The first two bytes of this replacement instruction sequence generate a software interrupt, the base number of which may be specified by the user at the time the RXE conversion program is run. The final byte of the instruction sequence, the Fix-up ID Byte (FUB), is decoded by the software interrupt handler in the RXE_BASE segment that is prepended to the start of the Data segments by the RXE conversion program. Depending on the length of the original instruction sequence, one or more NOP instructions may also be emitted by the RXE conversion program to maintain instruction alignment of the replacement and subsequent machine code sequences. The definitions of the fix-up ID bytes follow:

Fix-Up ID Bytes

Fix-up ID	**Description**
B0-B4	This number is an index into a 16-element array of Data segment values. An RXE interrupt handler uses this index value to retrieve an actual segment value from a table kept in the RXE_BASE segment.
B5-B7	For direct register accesses, this value specifies the destination register into which the RXE interrupt should load the specified segment value. These register specifications are decoded as follows 0 AX register 1 BX register 2 CX register 3 DX register 4 SI register 5 DI register 6 BP register 7 Indicates that the register (or constant) indirection is to be employed and that the destination of the segment value is a memory location rather than a CPU register. The RXE interrupt handler decodes the actual form of the memory reference by inspecting the modr/m byte which (after processing by RXE Convert) follows any imm8 or imm16 terms in the register indirect addressing expression.

The RXE software interrupt handlers are initialized at program run-time. The values stored in the RXE_BASE table are fixed up after the Data segments have been loaded into memory by DOS (which performs normal Data relative fix-ups). Once the interrupt handlers have been initialized and the RXE_BASE table has been fixed up, control is passed on to the actual entry point of the XIP application.

When an RXE software interrupt is generated by the XIP program during its normal course of operation, the RXE interrupt handler gains control and decodes the FUB (and possibly modr/m and displacement bytes) at the interrupt's return CS:IP location to determine the operation that needs to be performed to satisfy the Data relative fix-up. Once the appropriate action has been performed by the RXE interrupt handler it adjusts the return CS:IP and returns control to the application program with the Data relative fix-up effectively performed. The RXE base interrupt handler decodes the movement of immediate segment data into registers and memory. A second interrupt handler (for interrupt number RXE base+1) is used solely to push immediate segment values onto the stack.

The RXE conversion program only supports 80186 or earlier CPUs.

Within those instruction sets there are three classes of instructions that RXE Convert supports. The supported instructions are:

Supported Instruction Fix-Ups...

Mnemonic	**Op-code**
mov r16, imm16	{B8..BF} FIXUP
mov r/m16, imm16	C7 modr/m [disp] FIXUP
push imm16	68 FIXUP

where:

imm16 is an immediate (constant) segment value in Code

r16 is a destination register

r/m16 is either a destination register or an address expression

modr/m is the CPU modr/m byte which defines the current r/m16 address expression

disp is an optional 8-bit or 16-bit term in the address expression as specified by modr/m

FIXUP is the segment's paragraph address injected into the Code segment by the linker

The Intel CPU manuals are the standard reference for how to decode the machine code emitted by compilers. Please refer to those manuals for more information on these instruction sequences.

Each of the above instructions is similar in one regard they all take a 16-bit immediate value as the source operand. This is the only operation that may be meaningfully associated with a fix-up. There are several other instructions that also act on imm16 data and which could be found in the instruction stream, which is terminated at a fix-up location. The imm16 instructions that are NOT supported by RXE follow:

UnSupported Instruction Fix-Ups...

Mnemonic	**Op-code**
mov sp, imm16	BC imm16
mov sp, imm16	C7 C4 imm16
CS:	2E
SS:	36
DS:	3E
ES:	26

```
imul  r/m16, imm16        69 modr/m [disp] FIXUP
test  r/m16, imm16        F7 modr/m [disp] FIXUP
add     r/m16, imm16      81 modr/m [disp] FIXUP
or      r/m16, imm16      81 modr/m [disp] FIXUP
adc     r/m16, imm16      81 modr/m [disp] FIXUP
sbb     r/m16, imm16      81 modr/m [disp] FIXUP
and     r/m16, imm16      81 modr/m [disp] FIXUP
sub     r/m16, imm16      81 modr/m [disp] FIXUP
xor     r/m16, imm16      81 modr/m [disp] FIXUP
cmp     r/m16, imm16      81 modr/m [disp] FIXUP
add     ax, imm16         05 FIXUP
or      ax, imm16         0D FIXUP
adc     ax, imm16         15 FIXUP
sbb     ax, imm16         1D FIXUP
and     ax, imm16         25 FIXUP
sub     ax, imm16         2D FIXUP
xor     ax, imm16         35 FIXUP
cmp     ax, imm16         3D FIXUP
test    ax, imm16         A9 FIXUP
retn    imm16             C2 FIXUP
retf  imm16               CA FIXUP
enter imm16, imm8         C8 FIXUP
dw    @Data               @Data
```

where:

Imm16 is an immediate (constant) segment value in Code

r16 is a destination register

r/m16 is either a destination register or an address expression

Modr/m is the CPU modr/m byte which defines the current r/m16 address expression

Disp is an optional 8-bit or 16-bit term in the address expression as specified by modr/m

FIXUP is the segment's paragraph address injected into the Code segment by the linker

NOTES:

1) The "mov sp, imm16" sequence is specifically unsupported.

2) Segment overrides are not detectable and thus not supported by RXE. Any possible occurrence of a segment override will cause RXE Convert to generate an unsupported segment override warning describing the op-code’s location

3) The reg field of the modr/m byte is used to extend the meaning of all opcodes of value 81 and so differentiate one instruction from another. All other symbols are as above.

4) The declaration of data that will cause a fix-up in a code segment is specifically unsupported.

Should the RXE Convert program discover any of the unsupported opcodes above (except segment overrides) it will, if it cannot reasonably interpret them as part of a supported instruction sequence, display a message describing the nature and location of the instruction and abort. In fact, any sequence of bytes (such as is often encountered when data declarations appear in a Code segment) that cannot be reasonably decoded as a supported instruction sequence will cause the RXE Convert program to display a message and abort.

While the relative number of unsupported instructions seems large, years of practical experience indicate that the instructions that compilers emit to deal with fix-up values are almost exclusively direct register moves. Support for other cases have been added as required.

Handling Ambiguous Instructions

A more likely problem than that of an unsupported instruction sequence is that of an ambiguous instruction sequence. The reason ambiguous cases exist is that the RXE conversion program can only know the location of the fix-up value, not the beginning of the instruction sequence. The .EXE file header contains absolutely no information regarding instruction alignment as no such information is needed then the program is fully relocatable. To understand the significance of ambiguous instruction sequences let us consider the following assembly language examples:

```
mov   al,   C7H
mov   [si], SEG Data
```

If we assume the paragraph address of the Data segment is 0B7A0 then these assembly language instructions would cause the assembler/linker to emit the following sequence of bytes:

```
Address     -4 -3 -2 -1 0
Data        B0 C7 C7 04 A0 B7
```

The address values shown in the table above are relative to the start of the fix-up in the opcode sequence. As described before, the C7 opcode is generated by the supported assembly language instruction "mov r/m16, imm16". Note that there are two occurrences of the C7 opcode present in the byte sequence above and that both of those values appear within the maximum four-byte instruction length of that instruction. RXE Convert is faced with the dilemma of resolving the two choices. The possible interpretations of the above byte sequence are:

```
mov   di, A004
      -or-
mov   [si], B7A0
```

The key to the resolution of this problem is to note that only the latter interpretation will align the start of the required immediate data with the actual location of the fix-up in the Code segment and so the first case is clearly wrong. RXE Convert will correctly choose the latter case in this example because it has enough data to make a correct determination. Now consider this alternate example:

```
mov   [bx+si+68], SEG Data
```

The output from the assembler/linker in this case would be the following byte sequence:

```
Address     -3 -2 -1 0
Data        C7 40 68 A0 B7
```

In this case RXE Convert is faced with choosing between the following two supported opcode sequences.

```
mov   [bx+si+68], SEG Data
```

```
          - or -
      push  SEG Data
```

Both of these instructions align the fix-up properly and both decode as supported instruction sequences by RXE. There is no way for RXE Convert to tell which is the correct choice. It so happens that because RXE Convert processes single byte opcodes first, it will wrongly choose the latter choice in this case.

Because RXE Convert cannot guarantee it will always choose the right instruction sequence in ambiguous cases, warnings, describing the location of the code sequence in the .EXE file, will be generated whenever an ambiguous instruction sequence is detected. The following classes of ambiguous instructions produce warnings in RXE Convert:

A) unsupported instructions within a longer supported instruction sequence

B) supported instructions within a longer supported instruction sequence

C) supported instructions within a longer unsupported instruction sequence

D) possible segment overrides

NOTES:

1) Any purely unsupported instruction sequence, nested or not, will cause RXE Convert to display an error message and abort.

2) Any unclassifiable instruction sequence (one that can not be classified as supported or unsupported) will cause RXE Convert to display an error message and abort.

3) The declaration of a variable in a Code segment which causes a Data relative fix-up may or may not be detectable by RXE Convert and/or RXE Verify.

As illustrated, RXE Convert's warnings regarding ambiguous instruction sequences may or may not be benign. It is the users responsibility to investigate and validate such instruction sequences and determine the validity or the invalidity of the RXE Convert output.

EXE Header Description

When the .EXE to .RXE conversion process has been completed, all entries in the fix-up list, except for those that point to Data fix-ups within the Data segments, will have been removed from the .EXE header.

The RXE Convert program reduces the fix-up list by updating the number of fix-up entries remaining in the fix-up list, but does not change the size of .EXE header itself. In a large .RXE program there may be several kilobytes of unused space remaining in the .RXE header that previously contained fix-ups removed during the conversion process.

To complete the .EXE to .RXE conversion process, a number of fields in the .EXE header of the .RXE file need to be updated. The primary fields of the .EXE file header are shown below.

```
struct exehdr {
  unsigned short sig;     /* MS-LINK's signature == 4D5A hex */
  unsigned short imgmod;  /* image length mod 512 (bytes) */
  unsigned short imglen;  /* image length in 512-byte pages (including hdr) */
  unsigned short nreloc;  /* Number of relocation table items */
  unsigned short hdrlen;  /* Size of header in paragraphs */
  unsigned short minpar;  /* Minimum paragraphs required above loaded program */
  unsigned short maxpar;  /* Maximum paragraphs required above loaded program */
  unsigned short ssdsp;   /* Displacement of SS within module (paragraphs) */
  unsigned short spofs;   /* Offset to be put in SP when executing */
  unsigned short cksum;   /* Neg. sum of all words in file, ignoring overflow */
  unsigned short ipofs;   /* Offset to be put in IP when executing */
  unsigned short csdsp;   /* Displacement of CS within module (paragraphs) */
  unsigned short reldsp;  /* Displacement of 1st reloc item in file (bytes) */
  unsigned short ovlnum;  /* Overlay no. (0 for resident part) */
};
```

RXE Convert updates the following fields of the .RXE file's .EXE header:

imgmod	updated to reflect insertion of RXE_BASE segment
imglen	updated to reflect insertion of RXE_BASE segment
nreloc	updated to reflect remaining Data fix-ups in Data
hdrlen	updated to include the Code segments that run in ROM
ssdsp	updated to include the size of the RXE_BASE segment
ipofs	updated to point to the RXE init code as described
above	csdsp updated to point to the RXE init code as describe above

RXE Header Description

RXE Convert also fills in a data structure known as the RXE_BASE structure defined below.

```
struct rxe_base {
  long rxe_sig;          /* X - RXE Sig is "XIP" */ unsigned
  rxe_ver;               /* X - Version of RXE_BASE */ unsigned
  cd_seg;                /* X - Start of Code Block in RXE Memory */
  unsigned ld_seg;           /* R - Segment of start of RXE in memory */
  unsigned int_no;           /* R - Interrupt used for RXE support */
  unsigned cd_cs;        /* R - Segment of start code of RXE in memory */
  unsigned cd_ip;        /* R - Offset of start code of RXE in memory */
  unsigned cd_len;           /* R - Length in paragraphs of RXE code block */
  unsigned cd_sum;           /* R - Check sum of RXE code block */
  unsigned rxe_base_len; /* R -  The length in para's of the RXE Block */
  unsigned rxe_fixup_table[16]; /* R - Fix-up table */
  char code[2];          /* X - Code of the RXE_BASE */
};
```

Each field description begins with either an 'X' (meaning the data is static) or an 'R' which means that RXE Convert fills in the field when the .RXE file is created. The fields in the RXE_BASE structure are filled in as follows:

cd_seg	start of program image in ROM (following .EXE header)
ld_seg	user specified start of .EXE header in ROM
int_no	the (user assigned) RXE base interrupt number
cd_cs	application entry point segment in ROM
cd_ip	application entry point offset in ROM
cd_len	length of the Code block that executes in ROM
cd_sum	checksum of the above block
rxe_base_len	length of the RXE_BASE segment
rxe_fixup_table	paragraph address of Data segments
code	start of the RXE init code, followed by the interrupt handlers

RXE Convert Operation

In summary, the RXE conversion program performs a number of operations on a standard DOS executable to allow it to remain partially in ROM and so maximizing available memory. The steps in this process include:

1) Go through the .EXE file and analyze all fix-ups that appear in the .EXE file header. The action taken for each fix-up depends on the location of the fix-up and the location of the program element it references. The three cases are:

    A) For any Code reference, just use the ROM address specified on the RXE command line as the load address and fix-up those segment addresses with constant values. These fix-ups are removed from the .EXE header fix-up list.

    B) For each Data reference in Data, adjust the value to exclude the program code and include the RXE initialization code. DOS will fix-up all data segment references at load time.

    C) For any Data segment reference appearing in a Code segment, replace the CPU instructions that reference the data with special code that allows RXE to perform the fix-ups dynamically at run-time. These fix-ups are removed from the fix-up list in the .EXE header.

2) Prepend a special section of Code and Data known as the RXE_BASE segment to the Data. This special segment will be copied to RAM and initialized before the application to handle the dynamic fix-ups.

3) Adjust the .EXE header of the RXE so that:

    A) Only the RXE_BASE and the writable data segments are copied into RAM.

    B) Only fix-ups pertaining to Data in Data are performed by DOS.

    C) The RXE initialization code is the first code to execute and provides the handling of run-time fix-ups of Data references in Code.

    D) The changes are otherwise transparent to DOS and the application.

RXE Optimize Operation

Since .EXE header of the .RXE file must reside in ROM with the rest of the program image, the previously full list of fix-ups still uses a considerable amount of space. The RXE Optimize program can be used to shrink the size of the .RXE header and reduce the ROM requirements by removing unused space from the .EXE header of the .RXE file. The steps in the RXE optimization process include the following:

1) Compare the number of fix-up entries in the .RXE and .EXE file headers. The difference between these two values is the number of words (neglecting paragraph alignment) that can be removed from the .RXE header.

2) Adjust the differences in the fix-up counts for the required paragraph alignment of the start of the program image. Call this size, in bytes, the .RXE header delta.

3) Reduce the .RXE image size by copying the program image down over the free space in the .EXE header in .RXE file. A temporary file is used to create the optimized .RXE file.

4) Using the fix-up list in the .EXE header of the original .EXE file, locate each Code relative fix-up in the .RXE program image and subtract the .RXE header delta from the segment value. Data relative fix-ups are relocatable and not affected by this optimization, regardless of location.

5) Reduce the following fields in the RXE_BASE structure by the .RXE header delta:

   cd_seg
   cd_cs
   cd_ip

6) Adjust the following fields in the .RXE file's .EXE header by the .RXE header delta:

   hdrlen
   imgmod
   imglen

RXE Verify Operation

In order to make sure that the RXE Conversion process was successful, you can run Datalight's RXE verification tool. RXEVERFY performs a number of checks on the operation of both the RXE Conversion and RXE optimization programs. The verification steps taken by RXE verify include:

1) Verification that all parts of the RXE file that should not have been changed by RXE convert are, in fact, identical to the original .EXE file.

2) Recalculation and validation of each fix-up for both Code and Data to make sure the RXE file contains correct segment address data.   Each Data relative fix-up in Code is examined to make sure that the RXE file contains reasonable instruction sequences.

3) Recalculation and validation of the number and type of entries in the .EXE header fix-up list in the RXE file.

4) Recalculation and validation of entries in the .EXE file header.

5) Recalculation and validation of entries in the RXE_BASE structure.

6) Recalculation and validation of the RXE file checksum.

Errors and Warnings

When either the RXE conversion or verification program discovers a problem in the converted XIP program, a warning or error message is displayed which identifies the content and location of the problem.

In cases where ambiguous or unsupported instruction sequences are identified by RXE Convert, the following steps must be taken to validate the RXE output:

1) Make note of the location and type of problem noted in the RXE program's warning or error message.

2) Refer to the program's .MAP file to determine the module and the routine that contains the offending instruction sequence.

3) Recompile the offending module via assembly.

4) Locate the offending instruction sequences and isolate them in a test program that can be easily inspected to determine the affect of the RXE fix-up algorithms. The test program need not be effectively executable but must contain enough code before and after the offending instructions to determine if RXE Convert can make the correct decisions regarding those ambiguous instruction sequences.

5) Convert the test program from a .EXE to a .RXE and make sure that similar warnings are generated to ensure that the offending code is actually present in the test program.

6) Examine the RXE output in a debugger to determine if the proper instruction sequence was selected and that subsequent instruction alignment was maintained.

7) Modify the original program as required to avoid the offending instruction sequence generation. This may be most easily done by working with an intermediate assembly file.

NOTE: If any other types of problems are encountered using the RXE tools, contact Datalight for technical assistance.

RXE Verification

RXE verification consists of several hundred test cases that are designed to ensure that supported features are corrected properly and unsupported cases are correctly identified. Some test cases have been developed from error conditions found during the life of the product. A description of the test organization follows:

1. Verify successful conditions with and without optimized RXEs.

   1.1. Verify each supported opcode (described in Handling Runtime Fixups) with the following tests...

      1.1.1. Verify memory block before the target code fix-up instruction is not disturbed.

      1.1.2. Verify the proper start address of the instruction code fix-up located within the code segment. (CS)

      1.1.3. Verify the proper start address of the following instruction or memory block.

      1.1.4. Verify instruction fixup is correct in an ambiguous case.

      1.1.5. Verify that fixups are completed in the data area.

      1.1.6. Verify correct data fixups in the data area.

      1.1.7. Verify each possible segment override for each supported opcode.

   1.2. Verify each unsupported instruction is found by RXE_CVT.

   1.3. Verify each EXE header field is correctly modified.