A Simple Technique for Static Relocation of Absolute Machine Code

BY GARY A KILDALL
Digital Research
Pacific Grove, CA 93950

One principal difficulty with newly evolving computer technology is that software generation tools generally lag corresponding hardware facilities, thus forcing the software engineer to resort to outmoded techniques to produce software systems.

The purpose here is to present one area of difficulty–that of a static program relocation–and to offer a simple solution which can be applied to nearly any microcomputer software environment where relocation is not supported by the manufacturer.

The need for static relocation arises most often in a situation where the software systems must be reconfigured in the field. For example, data entry equipment manufacturers often provide a range of optional peripherals which can be attached to user's equipment as processing requirements change. Each peripheral usually requires a software "driver" which is device-specific, and interfaces the device to the operation environment.

A common approach to software reconfiguration is to arrange the individual translated peripheral drivers into distinct machine code modules which can be selectively brought together to form an integral system at the customer site. In order to perform the field reconfiguration, each module is translated so that it originates at location 0 in memory and, when it is brought together with other modules, it is placed at the next available memory location as the system is being constructed. All machine code elements which are location dependent must, of course, be altered to reflect the actual locations that the driver occupies. Generally, the elements which are affected are the addresses of branch destinations and data addresses. If the locations of the affected addresses in each module are known ahead of the system reconfiguration, the module can be placed anywhere in the final memory image.

Simple Static Relocation

The process of constructing an executable memory image from a set of relocatable modules, as described above, is called static relocation. Unfortunately, very few microcomputer manufacturers produce the address information with their translator output which is required for the relocation process. The method described below, however, can be applied to the output of most manufacturers' absolute translators to derive the necessary relocation information. In order to be specific, the Intel 8080 microcomputer is used in the discussion with the understanding that the concepts can be easily extended to differing architectures.

The Intel 8080 microcomputer has a 64K (65536 bytes) memory space which can be thought of as 256 "pages" of 256 bytes per page. Data and instructions are intermixed in this memory space, and are addressed with a 16-bit address operand which can be divided into an 8-bit (high-order) page address (0-255), and an 8-bit (low-order) address within a page. Typical 8080 instructions which can use these address operands are shown in Figure 1, where PA denotes the page address, and AWP denotes the address within a page. In general, a machine code memory image consists of instructions, instruction addresses, and data items. The instructions and data items are independent of the actual location at which the module finally resides. Further, only a subset of the instruction addresses are dependent upon the module location. That is, a load instruction may reference a buffer address which is fixed outside the relocatable module, in which case it does not change when the module is moved into position. If the address references a branch location or data item within the module, then the value of PA, AWP, or both, must be biased by fixed values, dependent upon the final position of the module in the resulting configuration.

MVI	A,	PA		Move immediate to A
MVI	C,	AWP		Move immediate to C
LXI	D,	AWP	PA	Load DE with address
JMP		AWP	PA	Jump to address

Figure 1. Typical 8Ø8Ø Instructions

A simpler form of relocation, called "page boundary relocation," is usually sufficient for field reconfiguration. In this case, the module is relocated to a page boundary so that only the page address (PA) need be changed to perform the relocation, since the address within a page (AWP) remains constant.

Page Boundary Relocation

In its simplest form, page boundary relocation can be accomplished by constructing two parallel memory images for each module. The first, called the "relative-0" image is created by translating the module for execution at location 0. The second, called the "relative-1" image is produced by translating the module for execution at page 1 (address 256). The relative-0 and relative-1 memory images can then be compared to determine the high-order address elements which must change when the module is moved to its final page boundary location. Figures 2a and 2b show a simple program segment assembled as relative-0 and relative-1 images. The differences in the machine code images are circled, and are thus the high-order addresses which must be biased when the module is moved. Figure 2c shows the same program segment assembled at page 5. Note that if the circled address fields in the ralative-0 image are biased by an amount 5 (corresponding to page boundary 5), they result in the proper values for the relocated program.

```
0000                org     000h  ;relative 0 assembly;
0000  3E00   start : mvi    a,d1 shr 8   ;page address to a
0002  0E0A          mvi     c,d1 and 0ffh ;address in page
0004  110A00        lxi     d,d1   ;full address to d, e
0007  C30000        jmp     start
             ;      data area
000A         d1:    ds      2      ;two unfilled
000C  00            db      0      ;one filled element
000D                end
```

```
Relative-0 Hex File:
:0A|0000|00|3E000E0A110A00C30000|C2
:01|000C|00|00F3
:00|0000|00|00
```

Figure 2a. Relative-0 Assembly

```
0100                org     100h  ;relative 1 assembly
0100  3E01   start: mvi     a,d1 shr 8   ;page address to a
0102  0E0A          mvi     c,d1 and 0ffh ;address in page
0104  110A01        lxi     d,d1   ;full address to d,e
0107  C30001        jmp     start
             ;      data area
010A         d1:    ds      2      ;two unfilled
010C  00            db      0      ;one filled element
010D                end
```

```
Relative-1 Hex File:
:0A|0100|00|3E010E0A110A01C30001|BE
:01|010C|00|00F2
:00|0000|00|00
```

Figure 2b. Relative-1 Assembly

```
0500                org     500h  ;assembly at page 5
0500  3E05   start: mvi     a,d1 shr 8   ;page address to a
0502  0E0A          mvi     c,d1 and 0ffh ;address in page
0504  110A05        lxi     d,d1   ;full address to d,e
0507  C30005        jmp     start
             ;      data area
             d1:    ds      2      ;two unfilled
050C  00            db      0      ;one filled element
050D                end
```

```
:0A|0500|00|3E050E0A110A05C30005|AE
:01|050C|00|00EE
:00|0000|00|00
```

Figure 2c. Assembly at Page 5

The program which actually performs the relocation process is a simple modification of an absolute loader. The translator output for an 8080 microcomputer is a "hex format" file, containing a sequence of absolute records which give a load address and byte values to be stored starting at the load address. The exact format of each record, shown in Figure 3, begins with a colon (:) followed immediately by a two digit record length (RL) and 4-digit load address (LA). The 2-digit record type (RT) is always zero for absolute records, and is followed by RL pairs of hexadecimal digits to be placed at LA through LA+RL–1 in memory. The record terminates with a pair of checksum digits: if the byte values (hexadecimal digit pairs) are summed, starting immediately after the colon, and, continuing through the end of the record, including the checksum byte, then the sum should be zero when computed with an 8-bit counter. The checksum byte is included as an error detection mechanism. The last record of a hex file is denoted by a record length of 00.

```
|:|nn|aaaa|tt|d1d2. . .dn|cc|

nn      —   record length 01-FF
aaaa    —   load address 0000-FFFF
tt      —   record type = 00
d1      —   data byte 1
d2      —   data byte 2
. . .
dn      —   data byte nn
cc      —   checksum byte
```

Figure 3. Hex File Format

An absolute loader reads each record of the hex file, and loads the byte values at the load address specified by LA for the next RL bytes, as shown in the algorithm of Figure 4. The notation used in this algorithm is that of Knuth [Kn.], where each step is labeled with a step name (A1. . . A16), followed by a comment describing the action of the step. The action itself is a series of assignments of expressions to variables, and conditional control transfers. The algorithm begins at step A1, and scans for the beginning colon for each record. When found, the algorithm reads the record length and, if zero, terminates the load operation. If the recrod length is not zero, the load address is read followed by the record type (which should be zero). The algorithm then loops between steps A6 and A12, reading successive bytes to memory while computing the checksum. When the entire record has been loaded, the final checksum byte is added, which should result in a zero value. Upon completion of the algorithm of Figure 4, the entire hex file has been read and loaded to an absolute location in memory.

The algorithm of Figure 5 is a simple extension of the previous absolute loader, which reads two successive hex files. The first hex file is the relative-0 machine code, produced by translating the module for execution at location 0. The second hex file is the relative-1 machine code, resulting from the module translation when originated at location 256 (100 in hexadecimal). The first part of the algorithm, given by steps A1 through A16 is similar to that of Figure 4, except that the data is loaded to address LA+PG*256 which effectively moves the module to the page boundary given by PG rather than absolute address LA.

```
Note:  nextchar reads the next ASCII character
       nextbyte reads the next pair of digits
       nextaddr reads the next pair of bytes
       CS is the checksum accumulator (8-bits)
       RL is the record length          (8-bits)
       LA is the load address           (16-bits)
       M[x] is memory location x        (8-bits)

A1  [scan for :]      if nextchar ≠ ":" go to A1
A2  [set checksum]    CS   := 0
A3  [get length]      RL   := nextbyte
A4  [last record?]    if RL = 0 go to A16
A5  [set address]     LA   := nextaddr
A6  [set type]        RT   := nextbyte
A7  [load bytes]      if RL = 0 go to A13
A8  [get byte]        b    := nextbyte
A9  [store byte]      M [LA]  := b
A10 [checksum]        CS   := CS + b
A11 [next addr]       LA   := LA + 1
A12 [count length]    RL   := RL – 1, go to A7
A13 [checksum]        CS   := CS + nextbyte
A14 [total ok? ]      if CS = 0 go to A1
A15 [check error]     halt, "checksum error"
A16 [normal end]      halt, "tape read ok"
```

Figure 4. Absolute Loader Algorithm

Upon reaching step A16, the module has been loaded into memory at page PG but is translated for execution at location 0 and thus would (most likely) execute improperly, since the high order branch and data addresses must be biased by an amount PG. Thus, steps R1 through R19 read the relative-1 hex file to determine the addresses which must change. These steps are similar to A1 through A16, except the input data is compared with memory for differences, rather than actually placed in memory. In step R5, the load address is read as before but, since the relative -1 machine code is biased by one page, the effective address must be reduced by 256 bytes. Step R9 compared the data loaded in the first phase with the data read in the second phase: if the data is the same, then the element is invariant in the relocation process. If the data differs, then it must have been due to the difference in the relative-0 and the relative-1 memory images. Further, this difference must be exactly 1 since differences occur only in the high-order address fields; otherwise an error occurs, and the module cannot be relocated. When a relocatable element is found, the original value loaded and relocated in phase 1 must be biased by an amount PG in step R11. Upon completion of the second phase, the algorithm halts at step R19 with the high order addresses altered by the proper amount in the relocated module. Note that the algorithm given in Figure 5, when applied to the relative-0 file of Figure 2a, followed by the relative-1 file of Figure 2b, produces the relocated machine code of Figure 2c, when page boundary PG=5 is used.

```
Note:  nextchar, nextbyte, nextaddr,
       CS, RL, LA, and M are identical to
       Figure 4. PG is the page number
       where the relocated module will reside.

A1  [scan for :]         if nextchar ≠ ":" go to A1
A2  [set checksum]       CS  := 0
A3  [get length]         RL  := nextbyte
A4  [last record?]       if RL = 0 go to A16
A5  [set address]        LA  := nextaddr
A6  [set type]           RT  := nextbyte
A7  [load bytes]         if RL = 0 go to A13
A8  [get byte]           b   := nextbyte
A9  [store byte]         M [LA + PG * 256] := b
A10 [checksum]           CS  := CS + b
A11 [next addr]          LA  := LA + 1
A12 [count length]       RL  := RL - 1, go to A7
A13 [checksum]           CS  := CS + nextbyte
A14 [total ok? ]         if CS = 0 go to A1
A15 [check error]        halt, "checksum error"
A16 [end rel-0]          go to R1

R1  [scan for :]         if nextchar ≠ ":" go to R1
R2  [set checksum]       CS  := 0
R3  [get length]         RL  := nextbyte
R4  [last record?]       if RL = 0 go to R19
R5  [set address]        LA  := nextaddr + 256 x (PG - 1)
R6  [set type]           RT  := nextbyte
R7  [record done?]       if RL = 0 go to R15
R8  [compare data]       b   := nextbyte
R9  [data same?]         if b = M [LA] go to R12
R10 [page diff 1?]       if b ≠ M [LA] + 1 go to R18
R11 [relocate]           M [LA]  := M [LA] + PG
R12 [checksum]           CS  := CS + b
R13 [next address]       LA  := LA + 1
R14 [count length]       RL  := RL - 1, go to R7
R15 [checksum]           CS  := CS + nextbyte
R16 [total ok?]          if CS = 0 go to R1
R17 [check error]        halt, "checksum error"
R18 [reloc error]        halt, "relocation error"
R19 [end rel-1]          halt, "relocation done"
```

Figure 5. Relocating Loader Algorithm

The algorithm of Figure 5 can be easily translated to an appropriate assembly or high-level language program to perform this relocation process.

The processing of Figure 5 can be used to produce a more compact form of the relocatable module by building a "bit vector" which tabulates the addresses which require relocation, rather than actually performing the relocation process. That is, in step R11 the address LA must be biased by an amount PG for proper execution when the module originates at address PG*256. Thus, on the first pass, the data can be read to memory and, upon completion of the pass, a bit vector is constructed which has one bit position for each address within the module. Before starting step R1, the entire bit vector is zeroed to indicate that no addresses require relocation. As the second phase processing proceeds, each relocation address determined in step R11 can be "marked" by setting the corresponding position of the bit vector. Upon completion of the algorithm, the bit vector contains zeroes in the positions corresponding to addresses which are invariant over the relocation, and ones in the positions which require biasing by an amount PG. The entire relocatable module can then be saved for later static relocation.

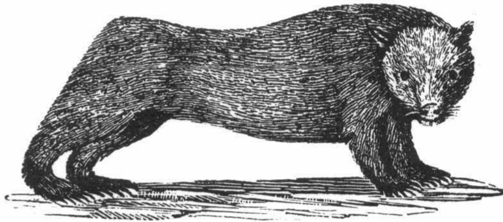
Given that the relative-0 memory image has been saved along with the relocation bit vector, the page boundary relocation can be simply accomplished by reading the memory image to its relocated page address PG. The bit vector is then read and processed: for each bit position which is set, the value PG must be added to the corresponding element which was previously loaded. Note that this extension to the basic algorithm of Figure 5 is included only for compact representation, and produces exactly the same memory image as the original algorithm.

A Case In Point

The following situation shows a case where page boundary relocation is useful. The CP/M operating system [Ki] is a simple small computer diskette based software system, which implements a file management and program loading facility for microcomputer development. The operating system is arranged as a set of modules which are loaded into memory when the computer system is started. User programs are then loaded into memory from the diskette and, because of memory constraints, must overlay non-essential portions of the CP/M system to reclaim storage for program and data areas. In order to allow these areas of memory to be reclaimed, the CP/M system is loaded into the high addresses of the memory space, and the user programs are loaded into the low addresses. Thus, the user programs can overlay the high addresses of memory when necessary and, upon completion, cause the CP/M system to be brought back from the diskette for the next operation.

Given that relocation is not supported by the manufacturer, this memory organization presents a fundamental difficulty: each CP/M operating system must be tied directly to the memory size. If the user of CP/M owns a computer system with 16K bytes of memory then a 16K version of CP/M must be supplied. If the user adds memory to enhance system capabilities, a different version of CP/M must be supplied to support the larger memory space.

In order to overcome this difficulty, the CP/M system can be reconfigured in the field to accomodate the increased memory using the page boundary relocation technique described above. In particular, each user receives a 16K version of CP/M (the smallest amount of memory which is useful for CP/M operation), along with a program which implements the reconfiguration. The user may optionally execute the program which rebuilds the CP/M system, according to the existing memory size, and places the relocated memory image back on the diskette, ready for subsequent loading.



The CP/M debugger program, called the Dynamic Debugging Tool (DDT), also resides in the upper portions of memory so that it can co-exist with the programs under test. Again, the area in which DDT is loaded depends upon the current memory configuration, and thus page boundary relocation is performed each time the DDT program is brought into memory. The increased elapsed time for relocation of DDT is negligible when compared to an absolute load, as long as the bit vector technique of the previous section is used.

Restrictions

It should be noted that the technique described here is by no means a complete linking loader: no address resolution is provided between modules, and no load-time address arithmetic is allowed. Sets of modules which co-exist in an integral system must communicate through instruction and data addresses. Using the technique presented here, the communication must be performed through dedicated absolute addresses for data items. Further, instruction addresses must be established through a "root module" which contains a jump vector with vector elements for each possible module which could be configured in a final system.

Address arithmetic is often useful when combining modules. In the simple page boundary relocation described above, all address arithmetic must be performed at assembly or compile time, and must consist only of simple operations which involve a fixed positive or negative offset from a base address, or a shift or logical and operation which extracts the 8-bit page address of a full 16-bit address. A relocation error will occur, for example, if an 8-bit immediate operand instruction is obtained from a 7-bit right shift rather than an 8-bit right shift of an address quantity.

In spite of these shortcomings, the technique has particular advantages in being independent of a manufacturer's capabilities, whims, and fancies. All language processors must eventually produce an absolute memory image for execution on the target machine, and thus the relocation process presented here will continue to operate when new software tools are introduced.

Acknowledgements

The author would like to point out that the techniques presented here, although useful, are most likely not original or particularly inventive. In fact, at least one individual, Bruce VanNatta of IMS Associates, San Leandro, California, has independently applied the methods to produce relocatable PL/M programs. There are most certainly many other software designers who have approached the relocation problem in a similar fashion.

References

Kildall, G., *Microcomputer Software Design: A Checkpoint.* Proceedings of the Fall Joint Computer Conference, 1975.

Knuth, D., *The Art of Computer Programming. Volume I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1975.

FASTER DATA ENTRY FOR SC/MP

Dear Dr. Dobbs, *Received: 78 Jan 16*

Using the SC/MP kit (actually it is called Introkit this side of the Big Lake) with the keyboard addition I often cursed the program entry method as being somewhat slow. You have to press MEM, TERM, then enter your data, and finally press TERM again. That makes 5 keystrokes per byte where most systems only use 3, i.e. two for the hex digits and one carriage return. The enclosed program should solve this problem. All should be self-explanatory except for a few details:

1. The initial data display is always 00, the following (after first TERM) shows true data read from memory.
2. Assembler syntax is not standard but from a home-brewed 'cross' on an HP 9825A. The main difference is in hex notation – I use xxH where National uses X' or leading zero.
3. The program is relocatable as only PC-relative jumps are used (besides using SCMPKB which is assumed to start in 0000).

Best of all,

Erik Skovgaard
Nordlundsvej 10
DK-2650 Hvidovre
DENMARK


```
***** FSTKEY SCMPKB FAST DATA ENTRY ROUTINE *****

$$$$      Track 1  File #  6

  1                           ;ROUTINE FOR FAST ENTRY OF
  2                           ;PROGRAMS USING SCMPKB SYSTEM
  3                           ;JUST KEY IN BYTE AND PRESS TERM
  4                           ;ADDRESS IS THEN AUTO INCREMENTED
  5                           ;AFTER STORING THE BYTE.
  6                           ;ABORT KEY RETURNS CONTROL TO SCMPKB
  7                           ;PROGRAM ENTRY ADDR IS GIVEN IN LOC
  8                           ;0F15 (UPPER) AND 0F16 (LOWER)
  9                           ;
 10 0F30 C40F          LDI 0FH          ;P2=0F00
 11 0F32 36            XPAH P2
 12 0F33 C400          LDI 0
 13 0F35 32            XPAL P2
 14 0F36 C215          LD 15H(2)        ;GET ADDRESS
 15 0F38 CA0E          ST 0EH(2)
 16 0F3A C216          LD 16H(2)
 17 0F3C CA0C          ST 0CH(2)
 18 0F3E C400          LDI 0            ;CLEAR DATA
 19 0F40 CA0D          ST 0DH(2)
 20 0F42 900B          JMP "GODIS"
 21 0F44 C20D    NUM:  LD 0DH(2)
 22 0F46 1E            RR               ;SHIFT DIGIT
 23 0F47 1E            RR
 24 0F48 1E            RR
 25 0F49 1E            RR
 26 0F4A D4F0          ANI F0H          ;AND APPEND
 27 0F4C 58            ORE              ;NEW DIGIT
 28 0F4D CA0D          ST 0DH(2)        ;SAVE RESULT
 29 0F4F C401  GODIS:  LDI 1
 30 0F51 37            XPAH P3          ;P3=DISP ROUTINE
 31 0F52 C43F          LDI 3FH
 32 0F54 33            XPAL P3
 33 0F55 3F            XPPC P3          ;DISPLAY
 34 0F56 9002          JMP "CMD"        ;COMMAND RETURN
 35 0F58 90EA          JMP "NUM"        ;NUMBER RETURN
 36                           ;
 37                           ;COMMAND PROCESSING
 38                           ;ANY CMD EXCEPT ABT IS OK.
 39                           ;
 40 0F5A C20E    CMD:  LD 0EH(2)        ;P3=NEW ADDR
 41 0F5C 37            XPAH P3
 42 0F5D C20C          LD 0CH(2)
 43 0F5F 33            XPAL P3
 44 0F60 C20D          LD 0DH(2)
 45 0F62 CF01          ST @1(3)         ;STORE BYTE, INCR
 46 0F64 C300          LD 0(3)          ;READ NEXT BYTE
 47 0F66 CA0D          ST 0DH(2)
 48 0F68 37            XPAH P3          ;STORE NEW ADDR
 49 0F69 CA0E          ST 0EH(2)
 50 0F6B 33            XPAL P3
 51 0F6C CA0C          ST 0CH(2)
 52 0F6E 90DF          JMP "GODIS"      ;DISP NEW VALUES


  NUM = 0F44
GODIS = 0F4F
  CMD = 0F5A
```