the engineering and scientific software series

86/PC experts–PL/M-86

Using 86/PC in the Tektronix Environment

Caine, Farber & Gordon, Inc.

Warren Point International Limited

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure of the software described herein is governed by the terms of a license agreement or, in the absence of an agreement, is subject to restrictions stated in paragraph (b)(3)(B) of the Rights in Technical Data and Computer Software clause in DAR 7-104.9(a) or in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause in FAR 52.227-7013, as applicable.

Comments or questions relating to this manual or to the subject software are welcomed and should be addressed to:

Caine, Farber & Gordon, Inc.
1010 East Union Street
Pasadena, CA 91106
USA
Tel: (818) 449-3070
Telex: 295316 CFG UR

Warren Point International Limited
Babbage Road, Stevenage
Hertfordshire SG1 2EQ
ENGLAND
Tel: Stevenage (0438) 316311
Telex: 826255 DBDS G

ISBN 1-55714-009-X
Order Number: 9301-8

20 June 1984

CONTENTS

APPENDICES

FIGURES

1. LINKING 86PC USING TEKTRONIX TOOLS

86pc can generate Tektronix-compatible object files (LAS format). However, an application program frequently consists of several object files which must be linked together. In any case, the linked application program must be assigned physical addresses in the target system. This chapter addresses linking an application program and assigning it physical addresses; both operations are performed with the Tektronix link program.

1.1 DEFINITIONS

In this section, we define some terms used in other sections.

1.1.1 Addressability

X is said to be addressable from Y if the quantity

(address of X) - 16 * ((address of Y) / 16)

is no larger than 64K-1. Since the address space of the 8086 wraps around from 0FFFFFH to 0, X is considered to lie in the virtual address range [100000H, 10FFFFH] when X lies in [0, 00FFFFH] and Y lies in [0F0000H, 0FFFFFH]. Examples:

X	locations addressable from X
010000H	010000H-01FFFFH
0EAB95H	0EAB90H-0FAB8FH
0FDE80H	0FDE80H-0FFFFFH and 000000H-00DE7FH (virtual 0FDE80H-10DE7FH)

1.1.2 Combined segments

A class of segments C is said to be combined when no more than 64K of C segments are allowed to be present. All the C segments are conceptually combined into a single C segment; they must be addressable from a common location.

1.1.3 Separate segments

A class of segments C is said to be separate when more than 64K of C segments are allowed to be present (there need not actually be more than 64K). Each C segment is conceptually disjoint from every other C segment; they need not be addressable from a common location.

1.1.4 Segments separate from other segments

A class of segments C is said to be separate from another class of segments C' when there is no explicit requirement that C segments and C' segments be addressable from a common location. It is not required that the C segments or the C' segments themselves be separate.

1.1.5 Merged segments

A class of segments C is said to be merged with another class of segments C' when any data for a C segment is actually allocated within a corresponding C' segment. No C segments will actually appear.

1.1.6 Distinct segments

A class of segments C is said to be distinct when C segments are not merged with any other class of segments. C segments will appear in the object file.

1.2 SOME LINKER CONSIDERATIONS

The link program will enforce the following restrictions on the object files to be linked:

- A combined code segment may wrap around location 0; that is, it may occupy locations 0F8000H-0FFFFFH and 000000H-007FFFH. No other combined segment may wrap around location 0 (but uncombined segments may).

1.3 USING ICS TO CREATE A LINKER COMMAND FILE

The Tektronix ics program (and icsp, the ics prompter) may be used with object files produced by 86pc. This is particularly useful when a Pascal main program or the Pascal I/O libraries are used. The primary output of ics is a command file for link, the Tektronix linker.

1.3.1 Restrictions

The ics program will enforce the following restrictions on the object files to be linked:

- No more than 64K of code. Thus, the "medium" and "large" models may not conveniently be linked in this way.
- No more than 64K of constants + data + stack. Thus, the "large" model may not conveniently be linked in this way, and problems may even occur for the "compact" model.

1.3.2 Pascal main program and i/o

A Pascal main program object module resides in bjdrv.po, and two 86/PL support object modules reside in bjsup.86t and blackj.86t. The ics source file bj6.is contains

```
PASCAL_CONFIGURATION     Default Configuration
HARDWARE_CONFIGURATION   8086
INSTRUCTIONS_ROM         [00060H,07FFFH]
CONSTANTS_ROM            [08000H,08FFFH]
GLOBAL_VAR_RAM           [09000H,0D7FFH]
HEAP_STACK_RAM           [0D800H,0FFFFH]
RESET_MEMORY             NO
SERVICE_CALLS            DEFAULT
SOFTWARE_CONFIGURATION   bjdrv.po
MODULE                   bjsup.86t
MODULE                   blackj.86t
LIBRARY                  NONE
FILE_SUPPORT             DEFAULT
INTERRUPT_CONFIGURATION  NONE
RESTART_LABEL            PASCAL_BEGIN
END
```

which produces the object file bj6.io and the linker command file bj6.ic, containing

```
-O bj6.io
-D STKBASEQQ=0FFFFH
-D HEAPBASEQQ=0D800H
-D SVCLOCZZ=0FFF0H
-m INSTRQQ.ROM=060H-07FFFH
-m CONSTQQ.ROM=08000H-08FFFH
-m DATAQQ.RAM=09000H-0D7FFH
-m SRBVQQ.RAM=040H-05FH
-L class=INSTRQQ range INSTRQQ.ROM
-L class=CONSTQQ range CONSTQQ.ROM
-L class=DATAQQ range DATAQQ.RAM
-L class=SRBVQQ range SRBVQQ.RAM
-O bjdrv.po
-O bjsup.86t
-O blackj.86t
-D CODEBASEQQ=060H
-D DATABASEQQ=08000H
-O /lib/8086/pas.hiio.scsd
-O /lib/8086/pas.fp86.scsd
-O /lib/8086/pas.rts.scsd
-O /lib/8086/pas.err.scsd
-O /lib/8086/pas.posi.scsd
-O /lib/8086/pas.conv.scsd
-x PASCAL_BEGIN
```

The Tektronix linker may then be invoked with

```
link -o bj6 -c bj6.ic
```

1.3.3 No Pascal routines

An 86/PL main program object module resides in root.q, and two 86/PL support object modules reside in first.q and second.q. The ics source file root.is contains

```
PASCAL_CONFIGURATION     86pc example
HARDWARE_CONFIGURATION   8086
INSTRUCTIONS_ROM         [0E0000H,0E9FFFH]
CONSTANTS_ROM            [0EA000H,0EEFFFH]
GLOBAL_VAR_RAM           [0EF000H,0F0FFFH]
HEAP_STACK_RAM           [0F1000H,0F2FFFH]
RESET_MEMORY             YES
SERVICE_CALLS            NONE
SOFTWARE_CONFIGURATION   root.q
MODULE                   first.q
MODULE                   second.q
LIBRARY                  NONE
FILE_SUPPORT             NONE
INTERRUPT_CONFIGURATION  NONE
RESTART_LABEL            PASCAL_BEGIN
END
```

which produces the object file root.io and the linker command file root.ic, containing

```
-O root.io
-D STKBASEQQ=0F2FFFH
-D HEAPBASEQQ=0F1000H
-m INSTRQQ.ROM=0E0000H-0E9FFFH
-m CONSTQQ.ROM=0EA000H-0EEFFFH
-m DATAQQ.RAM=0EF000H-0F0FFFH
-L class=INSTRQQ range INSTRQQ.ROM
-L class=CONSTQQ range CONSTQQ.ROM
-L class=DATAQQ range DATAQQ.RAM
-O root.q
-O first.q
-O second.q
-D CODEBASEQQ=0E0000H
-D DATABASEQQ=0EA000H
-O /lib/8086/pas.fp86.scsd
-O /lib/8086/pas.rts.scsd
-O /lib/8086/pas.err.scsd
-O /lib/8086/pas.noio.scsd
-O /lib/8086/pas.conv.scsd
-x PASCAL_BEGIN
```

The command file should be modified as follows:

- The line -O root.io includes the object module which interfaces with the Pascal run-time system; it should be removed.

- The lines -O /lib/8086/pas.<various>.scsd include the Pascal run-time routine libraries; they may be removed.

- The line -x PASCAL BEGIN references the standard Pascal main program entry point; it should be removed. The address of the main program code in root.q will be used as the entry point.

This leaves the command file

```
-D STKBASEQQ=0F2FFFH
-D HEAPBASEQQ=0F1000H
-m INSTRQQ.ROM=0E0000H-0E9FFFH
-m CONSTQQ.ROM=0EA000H-0EEFFFH
-m DATAQQ.RAM=0EF000H-0F0FFFH
-L class=INSTRQQ range INSTRQQ.ROM
-L class=CONSTQQ range CONSTQQ.ROM
-L class=DATAQQ range DATAQQ.RAM
-O root.q
-O first.q
-O second.q
-D CODEBASEQQ=0E0000H
-D DATABASEQQ=0EA000H
```

1.4 WRITING A LINKER COMMAND FILE DIRECTLY

If an application contains no Pascal object modules, it may be simpler to write the linker command file directly than to go through ics. More seriously, any model of computation other than "small" (the default) may violate the requirements of ics, thus precluding its use.

1.4.1 Important symbols

The object modules produced by 86pc were designed to be compatible with Pascal object modules. Pascal uses several symbols which are defined at link-time to resolve certain references. Object files produced by 86pc must also use these symbols in most cases.

CODEBASEQQ This symbol is only used when code segments are combined. It then corresponds to the address of cgroup in Intel-format object files, the (paragraph-aligned) address relative to which code segment references are made.

DATABASEQQ This symbol is only used when data segments are combined. It then corresponds to the address of dgroup in Intel-format object files, the (paragraph-aligned) address relative to which data segment references are made.

HEAPBASEQQ This symbol is always required. It is the (paragraph-aligned) address of the stack segment in Intel-format object files.

STKBASEQQ This symbol is always required. It is the largest address corresponding to a byte in the stack (not the address of the first byte past the stack). This number should be odd.

ENDREL This is the Tektronix equivalent of the MEMORY array. The linker has virtually complete control over this symbol.

INSTRQQ This is the class name of all code segments produced by 86pc and Pascal. This is true with 86pc for all models of computation.

DATAQQ This is the class name of all data segments produced by 86pc and Pascal. This is true with 86pc for all models of computation.

CONSTQQ This is the class name of all constant segments produced by 86pc and Pascal. This is true with 86pc for all models of computation in which distinct constant segments appear. This class is undefined when constants are merged with code or data.

The value of each of CODEBASEQQ, DATABASEQQ and HEAPBASEQQ must be a multiple of 16 (the hex value must end in "0"); the linker may produce strange error messages if this is not true.

1.4.2 Addressability requirements

We here give addressability and symbol existence requirements for various models of compilation.

Separate code segments
CODEBASEQQ is not required.

Combined code segments
CODEBASEQQ is required. All code segments must be addressable from CODEBASEQQ. Procedures with combined code expect the 8086 CS register to contain CODEBASEQQ/16.

Distinct constant segments
DATABASEQQ is required. All constant segments must be addressable from DATABASEQQ.

Separate data segments
DATABASEQQ is not required.

Combined data segments
DATABASEQQ is required. All data segments must be addressable from DATABASEQQ. Procedures with combined data expect the 8086 DS register to contain DATABASEQQ/16.

Stack separate from data
STKBASEQQ and HEAPBASEQQ are required. STKBASEQQ must be addressable from HEAPBASEQQ. Procedures with stack separate from data expect the 8086 SS register to contain HEAPBASEQQ/16.

Stack not separate from data

DATABASEQQ, STKBASEQQ and HEAPBASEQQ are required. HEAPBASEQQ must be addressable from DATABASEQQ, and STKBASEQQ must be addressable from HEAPBASEQQ and DATABASEQQ. Procedures with stack not separate from data expect the 8086 SS register to contain DATABASEQQ/16.

1.4.3 An example ("small" model)

An application consists of an 86/PL main program object module residing in root.q, and two 86/PL support object modules residing in first.q and second.q. The application has < 2700H bytes of code, < 300H bytes of constants, < 1800H bytes of data, < 200H bytes of stack, and no references to the MEMORY array. The target machine will have ROM for instructions from locations 0-03FFFH, ROM for constants from 0F0000H-0F07FF, and RAM from locations 0F0800H-0F47FFH.

First, we define the link-time constants:

```
-D CODEBASEQQ=000400H
-D DATABASEQQ=0F0000H
-D HEAPBASEQQ=0F4000H
-D  STKBASEQQ=0F47FFH
```

Next, we define address ranges for the various classes:

```
-L class=INSTRQQ range 000400H-003FFFH
-L class=CONSTQQ range 0F0000H-0F07FFH
-L class=DATAQQ  range 0F0800H-0F3FFFH
```

The class names must be capitalized as shown. Since no memory range was split, it is not necessary to name the memory ranges. Finally, we define the object files to be linked:

```
-O root.q
-O first.q
-O second.q
```

This gives us the linker command file

```
-D CODEBASEQQ=000400H
-D DATABASEQQ=0F0000H
-D HEAPBASEQQ=0F4000H
-D  STKBASEQQ=0F47FFH
-L class=INSTRQQ range 000400H-003FFFH
-L class=CONSTQQ range 0F0000H-0F07FFH
-L class=DATAQQ  range 0F0800H-0F3FFFH
-O root.q
-O first.q
-O second.q
```

1.4.4 An example ("medium-large" model)

The model of computation described does not correspond to any of Intel's models, nor does it have an associated compiler keyword. This particular model has

- an arbitrary amount of code (separate code segments)
- at most 64K of data + constants (combined data segments; distinct, combined constant segments)
- at most 64K of stack (stack separate from data)

It is generated by the -Mcs compiler switch. Note that, in general, ics cannot be used to generate a linker command file for an application compiled in this model: there may be more than 64K of code, and as much as 128K of constants + data + stack.

An application consists of an 86/PL main program object module residing in main.q, three 86/PL support object modules residing in sub1.q, sub2.q and sub3.q, and a library of support routines residing in support.lib. The application requires a large amount of space for code, < 2800H bytes for constants, < 0C000H bytes for data, and < 4000H bytes for the stack (the application has several reentrant procedures, hence the large stack). The target machine will have the following memory layout:

ROM locations 0-01FFFH, 020000H-02FFFFH, 090000H-0BFFFFH, 0F0000H-0F7FFFH, 0FFE00H-0FFFFFH

RAM locations 088000H-08FFFFH, 0C0000H-0CEFFFH

(the ROM chip for locations 0-01FFFH contains a complete interrupt-processing system, written previously; the ROM chip for locations 0FFE00H-0FFFFFH will contain an intersegment jump to the main program; neither is actually available for the application itself).

First, we define the link-time constants:

```
-D DATABASEQQ=0BD000H
-D HEAPBASEQQ=088000H
-D  STKBASEQQ=08FFFFH
```

(CODEBASEQQ is not required; DATABASEQQ is in ROM because the constants must go in ROM). Next, since the memory range for code is split, we name the part of memory available to code:

```
-m instructions=020000H-02FFFFH 090000H-0BCFFFH 0F0000H-0F7FFFH
```

Next, we define address ranges for the various classes:

```
-L class=INSTRQQ range instructions
-L class=CONSTQQ range 0BD000H-0BFFFFH
-L class=DATAQQ  range 0C0000H-0CBFFFH
```

(the code segments have the class INSTRQQ, even though they're separate).

Finally, we define the object files to be linked:

```
-O main.q
-O sub1.q
-O sub2.q
-O sub3.q
-O support.lib
```

This gives us the linker command file

```
-D DATABASEQQ=0BD000H
-D HEAPBASEQQ=088000H
-D  STKBASEQQ=08FFFFH
-m instructions=020000H-02FFFFH 090000H-0BCFFFH 0F0000H-0F7FFFH
-L class=INSTRQQ range instructions
-L class=CONSTQQ range 0BD000H-0BFFFFH
-L class=DATAQQ  range 0C0000H-0CBFFFH
-O main.q
-O sub1.q
-O sub2.q
-O sub3.q
-O support.lib
```

2. USING 86PC TO INTERFACE TO THE 8540/8560

The 8086 ultimately communicates with the outside world through IN and OUT instructions. During program development, it is normally inconvenient to have specific i/o ports assigned and connected to the development system. Normally, i/o is performed through interface routines ("read", "write") which exist in two forms, one for the final stage (in which the appropriate IN and OUT instructions appear), and one for the development state (in which "magic" linkages to the debugging/operating system appear). This chapter addresses development-stage i/o on the Tektronix 8540.

2.1 THE PASCAL SOLUTION

Pascal has a runtime support library described in Section 7 of the Tektronix 8560 Pascal 8086/8088 Compiler Users Manual. This library contains i/o functions and a host of other functions as well. The 86pc compiler can access all of the runtime support library routines except those that expect parameters in specific registers (notably the long integer operations, DIVLQQ, LSHRQQ, MODLQQ, MULLQQ and SHLQQ); those routines which expect arguments which are not constructs of 86/PL (like sets) may require strange linkages. Thus, for example, an 86/PL "read" routine can simply call the Pascal "test for end-of-line", "read a character", and "read end-of-line" routines.

Although the Pascal runtime support library provides a convenient, powerful set of i/o routines, they currently have one drawback. They are designed for use in the normal Pascal model of computation, namely, the "small" model. This model has combined code segments, and expects the data, constant and stack segments to be addressable from DATABASEQQ (combined data; distinct, combined constants; stack not separate from data), and therefore makes all pointers to be two-byte offsets from the (fixed) value of DATABASEQQ.

The code requirement cannot easily be overridden (a solution exists, but it requires extra assembly-language code and some special considerations when generating the linker command file) (see also Section A.3). The data/constant/stack requirement can be relaxed somewhat; the following must be observed when passing an address (pointer) to a runtime library routine:

- a passed address must be made into a two-byte quantity (by using the offset$of operator, if necessary);
- a passed address may only reference a datum in a segment not separate from the current data segment.

(see also Section A.4).

2.2 THE SVC SOLUTION

A more general solution to the i/o problem requires the use of the 8540 SVC (service call) facilities, described in Section 6 of the Tektronix 8540 System Users Manual. Appendix B and Appendix C contain listings of some assembly-language and 86/PL procedures which may simplify the 86pc user's task of interfacing with the SVC facilities. This section describes their function and use.

The procedures may be classified as

- an initialization routine
- SVC function routines
- SVC interface routines
- four-byte utility routines
- SVC executors (for emulation modes 0, 1, and 2)

The SVC function and interface routines are written to use a single SVC (namely, SVC1); equally well, the SVC function and interface routines could be passed an SVC number (1-8 or 0-7) as an extra argument. The SVC executors will function properly in either case. The variable mode should be initialized to the desired emulation mode (0, 1, or 2).

The routines have all been written assuming four-byte pointers. One of the four-byte utility routines must be modified if pointers are only two bytes.

2.2.1 Initialization routine

The routine initialize$srbs initializes the SRB pointer vector in low memory. It should be called before any SVC function routine or SVC interface routine is called.

2.2.2 Four-byte utility routines

The four-byte utility routines interconvert between LAS-format and Intel-format 4-byte quantities.

An LAS-long is a four-byte integer; its most significant byte is stored at its lowest-addressed location, and its least significant byte is stored at its highest-addressed location. An Intel-dword is also a four-byte integer; its least significant byte is stored at its lowest-addressed location, and its most significant byte is stored at its highest-addressed location.

An LAS-pointer is a four-byte unsigned quantity; its most significant byte is stored at its lowest-addressed location, and its least significant byte is stored at its highest-addressed location (its most significant 12 bits will normally be zeroes). An Intel-pointer is a four-byte quantity, composed of a two-byte offset and a two-byte selector or frame; the offset is at the

lower-addressed locations. Note that an Intel-pointer has a unique representation as an LAS-pointer, but an LAS-pointer has 64K different representations as an Intel-pointer; the conversion from LAS-pointer to Intel-pointer returns the representation with an offset in the range 0-0fh.

intel$dword$tolaslong
returns the LAS-long value of an Intel-dword

intelptrtolasptr
returns the LAS-pointer value of an Intel-pointer

las$long$to$intel$dword
returns the Intel-dword value of an LAS-long (passed as a dword)

las$ptr$to$intel$ptr
returns the Intel-pointer value of an LAS-pointer (passed as a dword) (two versions, depending on the model of computation)

2.2.3 SVC function routines

The SVC function routines take arguments from the referencing 86/PL program. The routines could be rewritten to also accept an SVC number (a byte or word argument). The SVC function routines pass a function number and their arguments to an SVC interface routine, and optionally extract information from the appropriate SRB and return it. The SVC function routines are not absolutely necessary; an industrious programmer could define literals for the various function numbers, and just call the SVC interface routines.

Starred routines do not actually appear in the listing; their definition should be obvious from the other routines.

2.2.3.1 Zero-argument routines.

The following routines take no arguments.

svc$abort	1fh - abort program
svc$exit	1ah - exit program
svc$last$coni	11h - get last CONI character
svclogerror	09h - log error message (logs previous status)
svc$read$clock	16h - read program clock

2.2.3.2 One-argument routines (pointer).

The following routines take a single argument, a pointer, which points at a RETURN-terminated filespec, specifying a load file to load. A pointer value is returned, the transfer address. This address may not be in a form suitable for transfer use: (o, f), (o+16, f-1), (o+16*2, f-2), ... (o+16*4095, f-4095) all specify the same physical address, but the second value (the frame or selector part) is very important to the program.

svc$load$ovl	17h - load overlay

2.2.3.3 Two-argument routines (byte, pointer).

The following routines take two arguments. The first, a byte, is the channel number. The second, a pointer, points at a RETURN-terminated name or filespec.

	svc$assign$channel	10h - assign channel
	svc$create$file	90h - create file
	svc$open$for$read	30h - open for read
*	svc$open$for$update	70h - open for read or write
*	svc$open$for$write	50h - open for write

2.2.3.4 Two-argument routines (byte, dword).

The following routines take two arguments. The first, a byte, is the channel number. The second, a dword, is a file offset. All the routines return a dword value, the new file offset (after the seek).

	svc$seek$relto0	44h - seek to byte in file
*	svc$seek$reltoeof	64h - seek to byte in file relative to eof
*	svc$seek$reltohere	24h - seek relative to byte in file

2.2.3.5 Three-argument routines (byte, word, pointer).

The following routines take three arguments. The first, a byte, is the channel number. The second, a word, is a number of bytes or characters to read or write. The third, a pointer, points at a line or buffer. All the routines return a word value, the number of bytes or characters read or written.

	svc$read$asc$go	81h - read ascii and proceed
	svc$read$asc$wait	01h - read ascii and wait
*	svc$read$bin$go	41h - read binary and proceed
*	svc$read$bin$wait	c1h - read binary and wait
*	svc$rewrite$asc$go	a2h - rewrite ascii and proceed
*	svc$rewrite$asc$wait	22h - rewrite ascii and wait
*	svc$rewrite$bin$go	e2h - rewrite binary and proceed
*	svc$rewrite$bin$wait	62h - rewrite binary and wait
	svc$write$asc$go	82h - write ascii and proceed
	svc$write$asc$wait	02h - write ascii and wait
*	svc$write$bin$go	42h - write binary and proceed
*	svc$write$bin$wait	c2h - write binary and wait

2.2.3.6 Three-argument routines (word, word, pointer).

The following routines take three arguments. The first, a word, is the number (ordinal) of the desired parameter. The second, a word, is the maximum size of the parameter, in bytes. The third, a pointer, points at a line or buffer which will hold the argument.

*	svcgetcmd$parm	13h - get command line parameter
*	svcgetexec$parm	16h - get execution line parameter

2.2.4 SVC interface routines

The SVC interface routines accept an SVC function number and a number of other arguments. These routines set up an SRB and then call svcgo. Svcgo calls the assembly-language SVC executor appropriate to the selected emulation mode.

The following table lists the arguments for the various SVC interface routines.

svcgo	(no arguments)
svcx	function (byte)
svcxb	function (byte), buffer (pointer)
svcxcb	function (byte), channel (byte), buffer (pointer)
svcxclb	function (byte), channel (byte), length (word), buffer (pointer)
svcxd	function (byte), offset (dword)
svcxplb	function (byte), position (word), length (word), buffer (pointer)

2.2.5 SVC executors

The SVC sequence for emulation modes 0 and 1 requires an OUT instruction followed by one NOP instruction; the sequence for mode 2 requires an OUT instruction followed by two NOP instructions. Since the compiler cannot generate arbitrary NOP's, these procedures must be written in assembly language. The assembly-language routine svcall1 performs the SVC sequence for emulation modes 0 and 1; svcall2 performs the sequence for mode 2.

3. TEKTRONIX LINKER ERRORS ENCOUNTERED WITH 86/PL OBJECT

This chapter describes some of the common linker errors and the situations which provoke them. It assumes that fairly standard linker commands are used; users sufficiently sophisticated to use linker features not described in Chapter 1 are assumed to need no further instruction here.

3.1 SECTION NAMES

The compiler names sections in the following manner:

I.module the code section for module

C.module the constant section for module

D.module the data section for module

A.module the absolute section for module (not always present)

If the section name would be longer than 16 characters, the first 6 and the last 10 characters of the name are used.

3.2 TYPICAL ERRORS

link:100 (S) Name symbol in section section previously defined

Symbol is declared "public" in section. It was either declared "public" in some other module, or it is one of the class names (INSTRQQ, CONSTQQ, DATAQQ) or address symbols (CODEBASEQQ, DATABASEQQ, HEAPBASEQQ, STKBASEQQ).

link:110 (E) No memory allocated to section

The memory range assigned to the section's class of segments (code, constant, or data) is not large enough to hold all of them. Section is effectively still relocatable. The memory range for section's class should be increased, or section's size should be decreased.

link:114 (E) Absolute section section conflicts with -L switch

Section is absolute, it appeared in a -L command, and the address in the -L command doesn't match the address of section. Remove the offending -L command.

link:115 (E) Truncation error at address

A 16-bit address-like quantity does not fit in 16 bits. Address is the physical address of the 16-bit quantity. It is not the address of the referenced item (the target). Typically, it is the address of the offset

in an assembly-language instruction, which is 1-4 greater than the address of the first byte in the instruction. See Section 3.3 for a more detailed description.

link:118 (W) Transfer address undefined

The object modules did not include a main program, and no -x command appeared. This may be the desired situation.

link:119 (W) Processor changed from family-1 to family-2

Different Tektronix processors include different microprocessors in the same family. If both families contain the desired target microprocessor, this warning is innocuous.

link:125 (W) Reserved name symbol used incorrectly

This appears to occur when a user program defines ENDREL to be a public datum or procedure. The linker has reserved the definition of ENDREL to itself.

link:128 (E) Absolute or symbol file section section cannot be relocated

Section is an absolute section, and it appeared in a -L command. This message appears even if the address in the -L command is appropriate to section. Remove the offending -L command.

3.3 TRUNCATION ERRORS

A truncation error occurs when a 16-bit address-like quantity does not fit in 16 bits. The address in the error message is the physical address of the 16-bit address-like quantity. It is not the address of the referenced item (the target). Typically, it is the address of the offset in an assembly-language instruction, which is 1-4 greater than the address of the first byte in the instruction.

There are two general causes of truncation errors:

- a signed quantity (a displacement) processed by the linker is outside the range [-32K, 32K-1] ([0ffff8000h, 000007fffh])
- an unsigned quantity (an offset) processed by the linker is outside the range [0, 64K-1] ([000000000h, 00000ffffh])

Displacements occur with jump and call instructions. Offsets occur with addresses in the data or const segments.

The following specific things can cause truncation errors:

- Suppose a call instruction at address 01000h references a procedure at address 0f000h. The calculated displacement is (0f000h - 01000h) or 0e000h, which is outside the displacement range. However, this is a spurious error. The 8086 interprets a displacement of 0e000h as

-02000h, but it performs displacement arithmetic within a 16-bit register, making 01000h - 02000h = -01000h = 0f000h. The same rationale holds for a call instruction at 0f000h which references a procedure at address 01000h.

- Consider the linker commands

```
-m data.ram=07000H-16FFFH
-L class=DATAQQ range data.ram
-D DATABASEQQ=01000H
```

If more than 9000h (36K) of data is present, some of the data is not addressable from DATABASEQQ; any attempt to get a 2-byte address of such a datum will provoke a truncation error.

- Consider the linker commands

```
-m data.ram=09000H-0FFFFH
-L class=DATAQQ range data.ramf
-D DATABASEQQ=09000H
```

The -L command will provoke a warning message of the form

```
link:10 (W) Undefined memory
```

The linker then places any data segments in the next available memory, not necessarily in the data.ram area. A 2-byte reference to any datum in a data segment which falls in the range 0-8FFFH or 19000H-0FFFFFH will provoke a truncation error. Note that eliminating the warning message (by correctly typing the memory range name) also eliminates the truncation error.

- Consider an 86/PL source file define.p containing the statement

```
declare absolute$thing word public at (234h);
```

and an 86/PL source file reference.p containing the statements

```
declare absolute$thing word external;
declare thing word;
...
thing = absolute$thing;
```

If define.p and reference.p are compiled with combined data, the compiler assumes that all external data are addressable from DATABASEQQ. In this example, if DATABASEQQ has a value greater than 240h, the value of the offset (address of absolute$thing - DATABASEQQ) is negative, and the linker finds a truncation error.

Note that this particular problem can only occur when an external reference is made to an item declared to reside at an absolute address and the corresponding segment is not separate. A local reference may always be made to an item at an absolute address. Perhaps the simplest method of dealing with items at absolute addresses which must be shared among several object modules is to place all the absolute

definitions (declared local, not public) in a single file which appears in an include control in every file which needs any such absolute definition. file

Appendix D contains an example of determining the root cause of the truncation errors in a link invocation.

APPENDICES

A. CONSIDERATIONS FOR MIXED MODELS OF COMPUTATION

This chapter describes some of the things which must be kept in mind when it is necessary to compile different object modules with different models of computation, and then link and execute the result.

A.1 DEFINITIONS

In this chapter, several terms have very specific meanings:

- A constant is any variable initialized with the DATA attribute, and any "long constant".
- A reentrant variable is any uninitialized variable local to a procedure with the REENTRANT attribute.
- A normal variable is any uninitialized variable which is not a reentrant variable, and any variable initialized with the INITIAL attribute
- An item is public if it is defined with the public attribute in the current object module.
- An item is local if it does not have the public or external attribute (it is local to the current object module).
- An item is private if it is defined within the current procedure (local to the current procedure).

All 86/PL procedures are DS-preserving: the value in the 8086 DS register upon return is the same as the value upon entry. All main programs initialize DS upon entry.

A.2 DISTINCT MODELS OF COMPUTATION

The model of computation has a single code-space component (c), and five data-space components (d, s, m, p, r). The following table gives the different data-space components of the possible distinct models of computation, and the total memory sizes implied by each. Every specification may also have a code-space component. Note that several specifications may be equivalent, and that any data-space component forces the p component (-M and -Mc are the only models in which pointer variables are two bytes long).

-M 64K data+stack+constant+memory

-Mp 64K data+stack+constant+memory

-Ms 64K data+constant+memory, 64K stack (also -Msp)

-Mm 64K data+constant+stack, 64K memory (also -Mmp)

-Msm 64K data+constant, 64K stack, 64K memory (also -Msmp)

-Mr 64K data+stack+memory, constants merged with code (also -Mrp)

-Mrs 64K data+memory, 64K stack, constants merged with code (also -Mrsp)

-Mrm 64K data+stack, 64K memory, constants merged with code (also -Mrmp)

-Mrsm 64K data, 64K stack, 64K memory, constants merged with code (also -Mrsmp)

-Md 1024K data, 64K stack, 64K memory, constants merged with data (also -Md[s][m][p])

-Mdr 1024K data, 64K stack, 64K memory, constants merged with code (also -Mdr[s][m][p])

Small corresponds to -M; compact, to -Msm; medium, to -Mcp; and large, to -Mcdr.

A.3 CALLING RESTRICTIONS

The various models of computation impose various restrictions on what calls can actually be properly executed.

A.3.1 Code segment combination

- A procedure may call any other procedure in the current object module, whether compiled with separate or combined code segments.
- No procedure may call a procedure in a different object module compiled with the "other" kind of code segments (a procedure compiled with separate code segments may not call an external procedure compiled with combined code segments, and conversely).

The second restriction arises from the nature of the "call" and "return" instructions and from the resulting stack layout in the different models of computaton.

A.3.2 Data segment combination

- Any local procedure may call any other procedure in any object module whatsoever (subject to the code segment combination restrictions).
- Any public procedure compiled with combined data may call any other procedure in any object module whatsoever (subject to the code segment combination restrictions).

- No public procedure compiled with separate data may call any procedure in a different object module compiled with combined data; except that it may reference a reentrant procedure in a different object module which only references private constants (if constants are merged with code), arguments, and reentrant variables (it references no data via the 8086 DS register).

A.4 POINTER RESTRICTIONS

The various models of computation impose various restrictions on what indirect (based) references can actually be properly made.

A.4.1 Four-byte pointers

- Assuming that four-byte pointers are present in the model of computation, any procedure may reference the datum to which any four-byte pointer points (there are no restrictions on the use of four-byte pointers).

A.4.2 Two-byte pointers

A two-byte pointer may be a pointer variable if four-byte pointers are not present, a word variable, or a procedure argument.

A.4.2.1 Pointers to procedures.

- Any procedure a compiled with combined code may reference (indirectly call) procedure b via a two-byte pointer, provided that b was also compiled with combined code.

A.4.2.2 Pointers to non-procedures.

- Any procedure may reference a local or public normal variable via a two-byte pointer.
- A procedure compiled with separate data may not reference any external datum via a two-byte pointer.
- A procedure compiled without distinct constant segments may not reference any constant via a two-byte pointer.
- A procedure compiled with stack separate from data may not reference any reentrant variable via a two-byte pointer.
- A procedure compiled with memory separate from data may not reference any datum in the MEMORY array via a two-byte pointer.

B. ASSEMBLY-LANGUAGE ROUTINES FOR SVC'S

This section gives the Tektronix-assembler source for SVC routines for the 8086.

B.1 "NEAR" PROCEDURES

These are the "near" procedures to perform SVC's in emulation modes 0, 1, and 2 on the Tektronix 8540. The "near" procedures must be used when the model of computation has combined code segments (the compiler "M" switch has no "c" subswitch).

```
        section i.srbs,class=instrqq
        global  svcall1
        global  svcall2

neararg equ     4

;       These are the 'near' procedures to invoke an arbitrary SVC.
;       INVOKE ONLY WHEN THE 'M' SWITCH HAS NO 'c' SUBSWITCH.

svcall1 push    bp
        mov     bp,sp
        mov     dx,neararg[bp]  ; put proper port (argument) in DX
        out     dx,al
        nop                     ; one no-op for mode-0 and mode-1
        pop     bp
        ret     #2              ; intrasegment ('near') return

svcall2 push    bp
        mov     bp,sp
        mov     dx,neararg[bp]  ; put proper port (argument) in DX
        out     dx,al
        nop                     ; two no-op's for mode-2
        nop
        pop     bp
        ret     #2              ; intrasegment ('near') return

        end
```

B.2 "FAR" PROCEDURES

These are the "far" procedures to perform SVC's in emulation modes 0, 1, and 2 on the Tektronix 8540. The "far" procedures must be used when the model of computation has separate code segments (the compiler "M" switch has a "c" subswitch).

```
        section i.srbs,class=instrqq
        global  svcall1
        global  svcall2

fararg  equ     6

;       These are the 'far' procedures to invoke an arbitrary SVC.
;       INVOKE ONLY WHEN THE 'M' SWITCH HAS A 'c' SUBSWITCH.

svcall1 push    bp
        mov     bp,sp
        mov     dx,fararg[bp]   ; put proper port (argument) in DX
        out     dx,al
        nop                     ; one no-op for mode-0 and mode-1
        pop     bp
        rets    #2              ; intersegment ('far') return

svcall2 push    bp
        mov     bp,sp
        mov     dx,fararg[bp]   ; put proper port (argument) in DX
        out     dx,al
        nop                     ; two no-op's for mode-2
        nop
        pop     bp
        rets    #2              ; intersegment ('far') return

        end
```

C. 86/PL ROUTINES FOR SVC'S

This section gives the 86/PL source for SVC routines for the 8086.

```
svc$test:
        do;

        /*****************************************************************
         *****************************************************************
         *                                                               *
         *      This source assumes that pointers are 4-byte items.      *
         *                                                               *
         *      This code is suitable only for the 8086/8088!            *
         *                                                               *
         *****************************************************************
         *****************************************************************/

/*      'Index' is the svc we'll be using.  It's declared literally,
        although it could be passed through as an argument to the
        procedures.                                             */
        declare index literally '0';

        declare svc1 literally '0fff7h',
                svc2 literally '0fff6h',
                svc3 literally '0fff5h',
                svc4 literally '0fff4h',
                svc5 literally '0fff3h',
                svc6 literally '0fff2h',
                svc7 literally '0fff1h',
                svc8 literally '0fff0h';

        declare svc (8) word data (
                svc1, svc2, svc3, svc4, svc5, svc6, svc7, svc8);

        declare srb (8) structure (
                fn byte,
                chan byte,
                status byte,
                four byte,
                count (2) byte,
                lth (2) byte,
                bufp dword);

/*      "Mode" is the mode (0, 1, 2) of the SVC's.  It should correspond
        to the value selected with the debugger EM command.  It may be
        declared public to make it easier to find should it be necessary
        to modify its value at debug-time.                      */
        declare mode word initial (0);
```

```
svcall1:
        procedure (port) external;
        declare port word;
end svcall1;

svcall2:
        procedure (port) external;
        declare port word;
end svcall2;



intel$dword$to$las$long:
        procedure (p) dword;

/*      Convert Intel-dword P into an LAS-long          */

        declare p dword;
        declare q dword, b (4) byte at (@q);

        b(0) = high (high (p));
        b(1) = high (p);
        b(2) = high (low (p));
        b(3) = p;
        return q;

end intel$dword$to$las$long;



intel$ptr$to$las$ptr:
        procedure (p) dword;

/*      Convert Intel-pointer P into an LAS-pointer     */

        declare p pointer;

        declare a dword;
        declare w word, ws selector at (@w);

        ws = selector$of (p);
        a = w;
        return shl (a, 4) + offset$of (p);

end intel$ptr$to$las$ptr;
```

```
las$long$to$intel$dword:
        procedure (q) dword;

/*      Convert LAS-long Q into an Intel-dword          */

        declare q dword, b (4) byte at (@q);

        declare a dword;
        declare w word;

        high (w) = b(0);
        low (w)  = b(1);
        high (a) = w;
        high (w) = b(2);
        low (w)  = b(3);
        low (a)  = w;
        return a;

end las$long$to$intel$dword;


las$ptr$to$intel$ptr:
        procedure (q) pointer;

/*      Convert LAS-pointer Q into an Intel-pointer     */

        declare q dword, v (4) byte at (@q);

        declare a dword;        /*  only needed for 2-byte pointers     */
        declare p pointer;
        declare w word, ws selector at (@w);

        w = shr (las$long$to$intel$dword (q), 4);
        selector$of (p) = ws;
        offset$of (p) = v(3) and 0fh;
        return p;

/*      If 2-byte pointers are used, the body should be  */

/*      ws = selector$of (@w);
        a = w;
        a = las$long$to$intel$dword (q) - shl (a, 4);
        offset$of (p) = a;
        if (high (a) <> 0);
                do while 1;
                        w = w + 1;
                end;
        endif;
        return p;       */

/*      The "if" checks for an address not reachable from DS (error);
        arbitrary recovery may be done (this routine just loops).   */

end las$ptr$to$intel$ptr;
```

```
/*
 *  initialize$srbs
 *
 *      This routine must be called before any svc is used!
 */

initialize$srbs:
        procedure public;

        declare srb$vect (8) dword at (00400h);

        srb$vect(0) = intel$ptr$to$las$ptr (@srb(0));
        srb$vect(1) = intel$ptr$to$las$ptr (@srb(1));
        srb$vect(2) = intel$ptr$to$las$ptr (@srb(2));
        srb$vect(3) = intel$ptr$to$las$ptr (@srb(3));
        srb$vect(4) = intel$ptr$to$las$ptr (@srb(4));
        srb$vect(5) = intel$ptr$to$las$ptr (@srb(5));
        srb$vect(6) = intel$ptr$to$las$ptr (@srb(6));
        srb$vect(7) = intel$ptr$to$las$ptr (@srb(7));

end initialize$srbs;


svc$abort:
        procedure public;

        call svcx (1fh);

end svc$abort;


svc$exit:
        procedure public;

        call svcx (1ah);

end svc$exit;


svc$last$coni:
        procedure byte public;

        call svcx (1fh);
        return srb(index).count(1);

end svc$last$coni;
```

```
svc$log$error:
        procedure public;

        call svcx (09h);

end svc$log$error;



svc$read$clock:
        procedure word public;

        declare w word;

        call svcx (1fh);
        high(w) = srb(index).count(0);  /*  reverse of 8086 natural order!  */
        low(w)  = srb(index).count(1);
        return w;

end svc$read$clock;



svc$assign$channel:
        procedure (c, b) public;

        declare c byte, b pointer;

        call svcxcb (10h, c, b);

end svc$assign$channel;



svc$create$file:
        procedure (c, b) public;

        declare c byte, b pointer;

        call svcxcb (90h, c, b);

end svc$create$file;



svc$open$for$read:
        procedure (c, b) public;

        declare c byte, b pointer;

        call svcxcb (30h, c, b);

end svc$open$for$read;
```

```
svc$seek$rel$to$0:
        procedure (c, o) dword public;

        declare c byte, o dword;

        call svcxcd (44h, c, o);
        return las$long$to$intel$dword (srb(index).bufp);

end svc$seek$rel$to$0;


svc$load$ovl:
        procedure (b) pointer public;

        declare b pointer;

        declare d based * dword;

        call svcxb (17h, b);
        return las$ptr$to$intel$ptr ((@srb(index).count)->d);

end svc$load$ovl;


svc$read$asc$go:
        procedure (c, m, b) word public;

        declare c byte, m word, b pointer;

        declare w word;

        call svcxclb (81h, c, m, b);
        high(w) = srb(index).count(0);  /*  reverse of 8086 natural order!  */
        low(w)  = srb(index).count(1);
        return w;

end svc$read$asc$go;


svc$read$asc$wait:
        procedure (c, m, b) word public;

        declare c byte, m word, b pointer;

        declare w word;

        call svcxclb (01h, c, m, b);
        high(w) = srb(index).count(0);  /*  reverse of 8086 natural order!  */
        low(w)  = srb(index).count(1);
        return w;

end svc$read$asc$wait;
```

```
svc$write$asc$go:
        procedure (c, m, b) word public;

        declare c byte, m word, b pointer;

        declare w word;

        call svcxclb (82h, c, m, b);
        high(w) = srb(index).count(0);  /*  reverse of 8086 natural order!  */
        low(w)  = srb(index).count(1);
        return w;

end svc$write$asc$go;



svc$write$asc$wait:
        procedure (c, m, b) word public;

        declare c byte, m word, b pointer;

        declare w word;

        call svcxclb (02h, c, m, b);
        high(w) = srb(index).count(0);  /*  reverse of 8086 natural order!  */
        low(w)  = srb(index).count(1);
        return w;

end svc$write$asc$wait;



svcgo:
        procedure;

        declare a byte;

/*      The next line is not necessary; it simply transfers the function
        value through a register (AL).  It is useful during debugging on
        the 8540:  if the user sets a breakpoint on the "if (mode < 2);"
        statement, the function value is immediately visible in the
        register display (one need not find the proper address in the srb
        vector and execute a debugger D command).  It is most useful when
        index is a variable rather than a literal.                  */
        a = srb(index).fn;

        if (mode < 2);
                call svcall1 (svc(index));
        else;
                call svcall2 (svc(index));
        endif;

end svcgo;
```

```
svcx:
        procedure (fn);

        declare fn byte;

        srb(index).fn = fn;
        call svcgo;

end svcx;


svcxb:
        procedure (fn, b);

        declare fn byte, b pointer;

        srb(index).fn = fn;
        srb(index).bufp = intel$ptr$to$las$ptr (b);
        call svcgo;

end svcxb;


svcxcb:
        procedure (fn, c, b);

        declare fn byte, c byte, b pointer;

        srb(index).fn = fn;
        srb(index).chan = c;
        srb(index).bufp = intel$ptr$to$las$ptr (b);
        call svcgo;

end svcxcb;


svcxcd:
        procedure (fn, c, o);

        declare fn byte, c byte, o dword;

        srb(index).fn = fn;
        srb(index).chan = c;
        srb(index).bufp = intel$dword$to$las$long (o);
        call svcgo;

end svcxcd;
```

```
svcxclb:
        procedure (fn, c, m, b);

        declare fn byte, c byte, m word, b pointer;

        srb(index).fn = fn;
        srb(index).chan = c;
        srb(index).lth(0) = high (m);    /*  reverse of 8086 natural order!  */
        srb(index).lth(1) = m;
        srb(index).bufp = intel$ptr$to$las$ptr (b);
        call svcgo;

end svcxclb;



svcxplb:
        procedure (fn, x, l, b);

        declare fn byte, x word, l word, b pointer;

        srb(index).fn = fn;
        srb(index).count(0) = high(x);   /*  reverse of 8086 natural order!  */
        srb(index).count(1) = x;
        srb(index).lth(0) = high(l);     /*  reverse of 8086 natural order!  */
        srb(index).lth(1) = l;
        srb(index).bufp = intel$ptr$to$las$ptr (b);
        call svcgo;

end svcxplb;


end;
```

D. TRACKING DOWN TRUNCATION ERRORS

This chapter gives an example of tracking down the cause of linker truncation errors.

An 8086 application, bj.86t2, is composed of a Pascal driver program, several 86/PL support modules, and the Pascal support libraries. The 86/PL modules are all compiled with

```
86pc -L -t name.p
```

There is no -M switch, so this is the "small" model of computation; there are no optimization switches, so only default optimization is performed. The application is linked with the command

```
uP=8086; export uP; link -d -o bj.86t2 -c bj.ic2
```

(bj.ic2, the linker command file, appears in Fig. D.1). The linker responds with the errors in Fig. D.2. The warning message is innocuous. There are no other messages, and bj.86t2 (the load module) is not produced.

```
-O bj6.io
-D STKBASEQQ=0FFFFH
-D HEAPBASEQQ=0D800H
-D SVCLOCZZ=0FFF0H
-m INSTRQQ.ROM=060H-07FFFH
-m CONSTQQ.ROM=08000H-08FFFH
-m DATAQQ.RAM=09000H-0D7FFH
-m SRBVQQ.RAM=040H-05FH
-L class=INSTRQQ range INSTRQQ.ROM
-L class=CONSTQQ range CONSTQQ.ROM
-L class=DATAQQ range DATAQQ.RAM
-L class=SRBVQQ range SRBVQQ.RAM
-O bjdrv.po
-O bjsup.86t
-O blackj.86t
-D CODEBASEQQ=060H
-D DATABASEQQ=09000H
-O /lib/8086/pas.hiio.scsd
-O /lib/8086/pas.fp86.scsd
-O /lib/8086/pas.rts.scsd
-O /lib/8086/pas.err.scsd
-O /lib/8086/pas.posi.scsd
-O /lib/8086/pas.conv.scsd
-x PASCAL_BEGIN
```

Fig. D.1 Linker command file

```
link:119 (W) Processor changed from  8086/87/88/186
                                  to       8086/8088
link:115 (E) Truncation error at       96
link:115 (E) Truncation error at       B4
link:115 (E) Truncation error at       E1
link:115 (E) Truncation error at       F7
link:115 (E) Truncation error at     835E
(8 more messages in ascending address order)
link:115 (E) Truncation error at     83F0
link:115 (E) Truncation error at      348
link:115 (E) Truncation error at      394
link:115 (E) Truncation error at      3AF
link:115 (E) Truncation error at      432
(80 more messages in ascending address order)
link:115 (E) Truncation error at      F0A
```

Fig. D.2 Linker error messages

What causes the truncation errors? From the lines

```
-m INSTRQQ.ROM=060H-07FFFH
-m CONSTQQ.ROM=08000H-08FFFH
-L class=INSTRQQ range INSTRQQ.ROM
-L class=CONSTQQ range CONSTQQ.ROM
```

in Fig. D.1, we see that the truncation errors appear in code segments (instruction sections) and constant segments. We will attack some of the code segment errors.

Which modules provoke truncation errors? There being no load module, we must examine the input object modules in the order they were linked. From Fig. D.1, we determine that the module order is

```
bj6.io
bjdrv.po
bjsup.86t
blackj.86t
(libraries)
```

We can probably safely ignore the libraries for now. We enter the command

```
lstr -o -s bj6.io bjdrv.po bjsup.86t blackj.86t | grep ' S '
```

The lstr command extracts all the symbol information from the object files; the -o switch identifies each symbol by its file, and the -s switch appends the section length to section information lines. The grep command copies every input line containing ' S ' (the "section information" code) to its output. The pipe symbol (the vertical bar) indicates that the output of lstr is to be the input to grep. The output appears in Fig. D.3. In general, segments I.name are code segments, C.name are constant segments, and D.name

```
bj6.io: 0x00000000 S %BJ6IO 0x00000000
bj6.io: 0x00000000 S ICS.INSTR 0x00000032
bjdrv.po: 0x00000000 S C.BJ 0x00000022
bjdrv.po: 0x00000000 S D.BJ 0x000000A4
bjdrv.po: 0x00000000 S I.BJ 0x000000BC
bjsup.86t: 0x00000000 S C.SUPPORT 0x00000000
bjsup.86t: 0x00000000 S D.SUPPORT 0x00000026
bjsup.86t: 0x00000000 S I.SUPPORT 0x000001CC
blackj.86t: 0x00000000 S C.BLACKJACKBODY 0x000005CA
blackj.86t: 0x00000000 S D.BLACKJACKBODY 0x000004AE
blackj.86t: 0x00000000 S I.BLACKJACKBODY 0x00000BF8
```

Fig. D.3 Section information

are data segments. ICS.INSTR is also a code segment (INSTR is the clue). The code segments have been placed in the address range [60H, 7FFFH], in the specific memory address ranges

```
          ICS.INSTR  [ 60H, (( 60H +   32H - 1) =   91H)]
              I.BJ   [ 92H, (( 92H +  0BCH - 1) =  14DH)]
         I.SUPPORT   [14EH, ((14EH +  1CCH - 1) =  319H)]
I.BLACKJACKBODY      [31AH, ((31AH + 0BF8H - 1) = 0F11H)]
```

Had bj.86t2 been produced, we could have entered the command

```
lstr -s bj.86t2 | grep ' S '
```

The resulting output would look like Fig. D.3, but without file names (there is no -o switch), and with appropriate physical addresses rather than relocatable 0's for the segment addresses. This would have avoided a bit of hex arithmetic.

The first four code segment errors are in bjdrv.po, the Pascal driver; the remaining 85 code segment errors are in blackj.86t, an 86/PL support module. We choose to attack the 86/PL module.

We recompile blackj.p (the source for blackj.86t) with

```
86pc -l -a -t blackj.p >blackj.s
```

Observe that

- The model of computation in the recompilation must match that in the original compilation (since we want the "small" model, we use no -M switch).
- The optimization in the recompilation must match that in the original compilation (since we performed no explicit optimization originally, we use no optimization switches).

- The -l switch should appear, so that a statement-numbered listing is produced.
- The -a switch should appear, so that an assembly-language listing is produced.
- Standard output should be redirected to a file, so that the listings are retained (we redirect it to blackj.s).

The assembly listing uses relative addresses; the corresponding values for the errors are

phys	relative
348H	(348H - 31AH) = 2EH
394H	(394H - 31AH) = 7AH
3AFH	(3AFH - 31AH) = 95H
432H	(432H - 31AH) = 118H

Part of the assembly-language listing appears in Fig. D.4, with the offending addresses and relevant statement numbers underscored. Note that the address in a truncation error message is the address of the relocatable datum; it is neither the address of the instruction or statement containing the datum, nor the target address. The problem instructions load the 8086 AX register with the "group-relative offset" of the variable name. The "group-relative offset" of x is the difference of the physical address of x and the value of the base symbol for the group (DATABASEQQ for dgroup, and CODEBASEQQ for cgroup). The offset must lie in the range [0, OFFFFH] (it must be nonnegative). Why does the linker find the group-relative offset of name offensive?

Part of the statement-numbered listing appears in Fig. D.5, with relevant statement numbers and source statements underscored. Note that not every statement in the assembly-language listing had a statement-number comment; the reference to name may be a few statements further down in the listing.

Notice that name is in a constant segment (it is initialized with the data attribute). Why should taking the "group-relative" address of a constant cause a problem? The answer lies in the following lines from Fig. D.1.

```
-m CONSTQQ.ROM=08000H-08FFFH
-L class=CONSTQQ range CONSTQQ.ROM
-D DATABASEQQ=09000H
```

They indicate that the value of

((physical address of name) - (value of DATABASEQQ))

is negative! A value was assigned to DATABASEQQ which appeared correct for the data segments, but which ignored the fact that it was necessary for other segments to be addressable from DATABASEQQ. Changing

```
-D DATABASEQQ=09000H
```

to

```
0020                    getnum  PROC    NEAR
0020   55                       PUSH    BP
0021   8bec                     MOV     BP,SP
                        ; statement #174, line #166
0023 R f606000401               TEST    tracev,1h
0028   7503                     JNZ     $+5
002a   e90700                   JMP     @0
002d R b8ea01                   MOV     AX,offset dgroup:name
0030   50                       PUSH    AX
0031 X e80000                   CALL    trace
                                ...
                        ; statement #181, line #173
004b   8b5e04           @3:     MOV     BX,[BP].cp
                                ...
006f R f606000401       @4:     TEST    tracev,1h
0074   7503                     JNZ     $+5
0076   e90700                   JMP     @5
0079 R b8ea01                   MOV     AX,offset dgroup:name
007c   50                       PUSH    AX
007d X e80000                   CALL    untrace
                        ; statement #188, line #180
0080 R a1a004           @5:     MOV     AX,num
0083   5d                       POP     BP
0084   c20200                   RET     2h
0087                    getnum  ENDP

0087                    putnum  PROC    NEAR
0087   55                       PUSH    BP
0088   8bec                     MOV     BP,SP
                        ; statement #206, line #198
008a R f606000401               TEST    tracev,1h
008f   7503                     JNZ     $+5
0091   e90700                   JMP     @6
0094 R b8f201                   MOV     AX,offset dgroup:name
0097   50                       PUSH    AX
0098 X e80000                   CALL    trace
                                ...
                        ; statement #224, line #216
010d                    @12:
010d R f606000401       @10:    TEST    tracev,1h
0112   7503                     JNZ     $+5
0114   e90700                   JMP     @13
0117 R b8f201                   MOV     AX,offset dgroup:name
011a   50                       PUSH    AX
011b X e80000                   CALL    untrace
                        ; statement #228, line #220
011e   5d               @13:    POP     BP
011f   c20600                   RET     6h
0122                    putnum  ENDP
```

Fig. D.4 Partial assembly-language listing

```
164     156     2       getnum:
165     157     2               procedure (cp) word public;
...     ...                     ...
172     164     3               declare name (*) byte data ('getnum', eos);
173     165     3
174     166     3               if tracev;
175     167     3                       call trace (dot$4 name);
176     168     3               endif;
...     ...                     ...
181     173     3               do while (c - '0') <= 9;
182     174     4                       num = num * 10 + (c - '0');
183     175     4                       cp = cp + 1;
184     176     4               end;
185     177     3               if tracev;
186     178     3                       call untrace (dot$4 name);
187     179     3               endif;
...     ...                     ...
190     182     3       end getnum;
...     ...                     ...
194     186     2       putnum:
195     187     2               procedure (num, where, size) public;
...     ...                     ...
204     196     3               declare name (*) byte data ('putnum', eos);
205     197     3
206     198     3               if tracev;
207     199     3                       call trace (dot$4 name);
208     200     3               endif;
...     ...                     ...
224     216     3               if tracev;
225     217     3                       call untrace (dot$4 name);
226     218     3               endif;
227     219     3
228     220     3       end putnum;
```

Fig. D.5 Partial statement-numbered listing

```
-D DATABASEQQ=08000H
```

cures the problem.

INDEX

Caine, Farber & Gordon, Inc.

750 East Green Street . . . Pasadena, California 91101
(818) 449-3070 . . . Telex 295316 CFG UR