SPERRY

XENIX Development System

Release Notes

Personal Computer Library
UP-11483

This document was typeset with an IMAGEN® 8/300 Laser Printer.

Document Number: G-2-14-85-1.3/1.0

XENIX 1.3 Development System Release Notes

Release Notes
Release 1.3
XENIX®-86 3.0 for Sperry® Personal Computers
Development System

1. Preface

These notes pertain to the XENIX-86 Development System for Sperry Personal Computers. They contain notes on the software, the procedure for installing the software and a list of the files on the floppies in this package.

We are always pleased to hear of user's experience with our product, and recommendations of how it can be made even more useful. All written suggestions are given serious consideration.

2. Software Notes

This release is notably enhanced by the incorporation of the Microsoft "cmerge C compiler." These notes will enable you to best take advantage of this compiler.

2.1 cc(CP)

The cmerge compiler, **cc**(CP), supports several important features. It allows you to compile programs for both XENIX and DOS. It also has stronger typing restrictions than the 1.1 C compiler (e.g. pointer arithmetic is more stringent).

2.1.1 Code Generation

The most important difference between compiling programs for XENIX and DOS is the treatment of long words. The XENIX System requires "big-endian" word order for longs (high order word first in memory). DOS requires "little-endian" word order for longs (low order word first in memory).

The cmerge compiler, **cc**(CP), defaults to small model XENIX 8086 code generation (big-endian word order) and uses the "b" libraries (for example *Sblibc.a* rather than *Slibc.a*). The default configuration flag is **-Mb0**.

The default stack for programs is a variable size stack, starting at the top of a full 64K byte data segment. A full 64K of physical memory is alloted for combined stack and data. The stack grows down until it hits the data. Use the **-F** flag to specify a fixed stack size.

The libraries included in the 1.1, 1.2 and 1.3 XENIX Development Systems use big-endian word order.

The 1.3 XENIX Development System release does not support the ability to produce executable programs suitable for XENIX 286 systems. It will produce 286 DOS executables and 286 relinkable object files. 286 program generation will be fully supported in future releases.

2.1.2 DOS Development Files

DOS executable files are created by using the **-dos** flag with **cc**(CP). The **-dos** flag causes the compiler to produce little-endian word order, use the DOS libraries and invoke the DOS linker, **dosld**(CP), instead of **ld**(CP). The DOS libraries are the "d" libraries, such as *Sdlibc.a*, found in */usr/lib/dos*.

Refer to XENIX *Programmer's Guide* (Chapter 10 "XENIX to MS-DOS: A Cross Development System," Appendix C "A Common Library System for XENIX and MS-DOS" and **dosld**(CP)) for more information on using XENIX to create programs suitable for DOS systems.

2.1.3 Large Model Program Generation

This release does not support large model 8086 program generation for XENIX. This will be supported in future releases. Large model is supported for DOS program generation.

2.1.4 Relinking Relocatable Files

The XENIX-86 1.2 and 1.3 Development System C compiler supports longer variable names than the previous C compilers. This feature may be suppressed by specifying the **-nl 8** flag on the command line to **cc**, for example:

```
cc -nl 8 *.o -o newprog
```

This is needed when using object code files (*.o* files) created by versions of the compiler previous to 1.2.

2.1.5 Assembly Language Generation

The **-S** and **-L** options to **cc**(CP), which generate assembly language source listings, do not function properly.

2.2 Link Kit

A Link Kit is provided with the XENIX 1.3 Operating System. The necessary files are on the Operating System floppy labelled "Link Kit floppy".

In order to save disk space, we recommend that this floppy not be installed unless required. The Link Kit may be installed at any time by using the */etc/install* command (see **install**(M)). Login as root (super-user) to remove the Link Kit once you are finished using it (or if it was inadvertently installed). Remove the directory */usr/sys* by (CAREFULLY!) typing:

```
rm -rf /usr/sys
```

The Link Kit enables you to add device drivers to your system. Additional device drivers are necessary to run non-supported peripheral devices. Refer to the two chapters in the XENIX *Programmer's Guide* for explanations on writing device drivers. They are "Writing Device Drivers" and "Sample Device Drivers".

After the device driver is written and compiled it is necessary to relink the XENIX kernel, including the new device driver, allowing XENIX to make use of the device.

Once you have a completed device driver (*driver.o*) and a correct configuration table (*c.o*) you need to relink the XENIX kernel. Copy the *driver.o* file and the *c.o* file to the directory */usr/sys/conf*. Edit the */usr/sys/conf/makefile* to include the name of the driver file. Add the name of the driver file to the line in the section "xenix.modules":

CONF=

You can then **make**(CP) a *xenix* or *xenix.sml*. by typing:

make xenix

or

make xenix.sml

The *makefile* tests for the existence of *xenix* or *xenix.sml*, backs it up, links a new *xenix* including the new device driver, and makes a new *namelist* file (see **nm**(CP)).

Note

Before using the shell script **make xenix** you MUST have '.' included in your PATH. If '.' is not included, */usr/sys/conf/rkseg* is not executed. The resulting kernel, when booted, will show nothing but a blank screen.

As super-user type the command:

PATH=.:$PATH

to temporarily include '.'.

Create a 'bootable xenix floppy' to test the new xenix kernel you have created. Insert a formatted floppy in drive 0 and type:

make boot

while in the directory */usr/sys/conf*. When you are finished, reboot the system from the floppy with your new kernel on it. Check the XENIX *Installation Guide* Chapter 2 "Installation Procedure" for instructions on booting the sytem from a floppy disk.

Now that you have a new, tested *xenix* kernel, you may install it onto the hard disk.

Note

Do not install a *xenix* on the hard disk until it is fully tested using a bootable floppy.

Use the shell script */usr/sys/conf/hdinstall* to install the new *xenix* on the hard disk. **hdinstall** can be invoked by typing:

```
/usr/sys/conf/hdinstall
```

or, while in the directory */usr/sys/conf*, type:

```
make hdinstall
```

To install a *xenix.sml* you must first move *xenix.sml* to *xenix*, as **hdinstall** installs the file */usr/sys/conf/xenix* on the hard disk.

3. Known Bugs

3.1 adb(CP)

The program debugger, **adb**(CP), does not single-step system calls. Subprocess control does not work on medium model programs.

3.2 as(CP)

as(CP) does not assemble the assembly code output from the C compiler **cc**(CP) when **cc** is given the **-S** flag. Future releases will allow C compiler code to be assembled.

Note

Assembly language programs of sufficient length will not assemble correctly (using */bin/as*). If this is the case, we recommend using */bin/as_1.1* with its different syntax or rewriting the program in C.

---

as_1.1 is included with this release. The manual page **as_1.1**(CP) and an alternate XENIX *Programmer's Guide* Chapter 7 "As_1.1: An Assembler" are included with these *Release Notes*.

3.3 ctime(S)

The **ctime**(S) library function needs to have the XENIX library *libx* included when linked. Specify the library on the **cc**(CP) line, for example:

```
cc -o prog *.o -lx
```

3.4 config(CP) and master(F)

master(F), the device information table and **config**(CP), which configures a XENIX system are not included with this release.

3.5 prof(CP)

The command **prof**(CP) is included with this release. However, if **cc**(CP) is used to compile a program using the -p option and **prof** is later run, you will get an error message:

```
No time accumulated
```

This is because the 1.3 XENIX kernel does not support profiling. Profiling will be supported in a later release.

3.6 signal(S)

The *function address* value given as a *func* (prescribing the action **signal** is to execute) must be an even address. **Signal** treats odd address as **SIG_IGN** (causing the process to ignore the signal.)

In XENIX-86 3.0 release 1.2 **signal** catching was not handled correctly. This problem has been fixed. An example of this incorrect behavior was manifested by **sleep**(S) not working.

3.7 Casting Function Calls

Do not cast function calls to **void**. The compiler does not correctly handle this cast. This will be fixed in a future release.

3.8 Infinite Spill Error

The error

```
Compiler Error (internal):
infinite spill
```

is provoked by excessive use of pointers and indexes. This will be fixed in a future release.

The line number given in the error message contains the expression causing the error. The error may be avoided by dividing the expression over several statements.

3.9 16 bit Left Shifts

The cmerge compiler generates incorrect code for expressions involving a left shift of 16 bits of an integer variable. Assign integer variables to long variables before attempting to left shift by 16 bits.

4. Documentation Notes

This section discusses changes and errors in the documentation.

4.1 New Documentation

The following new documentation is included with these *Release Notes*. Please insert them into your XENIX *Reference*.

XENIX *Programmer's Guide*
Chapter 7 "As_1.1: An Assembler"
alternate to current "As: An Assembler"
as_1.1(CP) alternate to current as(CP)
XENIX *Programmer's Reference*
sdget(S) replaces current sdget(S)

Both as and as_1.1 are available with the system. You can use either one, though as_1.1 is the newer version. A new Chapter 7 covering as_1.1 can be found in these release notes. A page describing as_1.1 to be inserted in the Commands(CP) section of the Programmer's Guide is also included in these release notes.

4.2 Appendix of Libraries

At the end of the (S) reference section in the *Programmer's Reference* is an appendix listing the XENIX system calls and libraries. The library listings include each of the functions included in that library.

The XENIX-86 1.3 Development System does not include large model libraries for the XENIX System. There are large model libraries for DOS.

4.3 read(S)

The command **rdchk**(S) should be listed under the section "See Also."

4.4 sdget(S)

The syntax for **sdget**, shown on the **sdget**(S) manual page, using the XENIX 1.3 kernel, should read **int size;** rather than **long size;**. A replacement manual page is included with these *Release Notes*.

5. Installation Notes

Note that you need the 1.3 (or equivalent) XENIX Operating System installed on your system in order to use the 1.3 XENIX Development System.

5.1 Disk Usage

The XENIX-86 1.3 Development System can be installed in two portions. If you intend to do DOS cross compilation, install the DOS development environment (linker, libraries and include files) in addition to the XENIX files. Otherwise you can install just the XENIX files. Floppies number 6 and 7 contain the DOS files and can be installed at a later time, if desired, by logging in as root (super-user) and using the */etc/install* utility explained later in this section or refer to the **install**(M) manual page.

The 1.3 XENIX Development System requires the following amount of free space (512 byte blocks) for installation.

3556	XENIX development files (Floppies 1-5)
824	DOS development files (Floppies 6-7)
4380	entire XENIX Development System

Use the **df**(C) command to check the amount of free disk space on your system. **df**(C) reports free disk space in 512 byte blocks.

5.2 Installation Procedure

To install the XENIX-86 1.3 Development System, follow this procedure:

1. Login as root (super-user) or enter system maintenance mode (single-user).

2. Insert the first Development System floppy into the floppy drive and enter the command:

   # /etc/install

3. The install utility will prompt:

   First floppy <y/n>?

   Enter 'y' and press RETURN.

4. The program will prompt you for each floppy. Insert the next Development System floppy and enter 'y' in response to the prompt:

   Next floppy <y,n>?

5. When you have installed the final Development System floppy, enter 'n' in response to the prompt:

   Next floppy <y,n>?

Note that some files may extend from one floppy to the next. In this case, the tar utility will prompt you in a slightly different fashion than the */etc/install* program.

'tar: please insert new volume, then press RETURN'

Insert the next floppy and press RETURN when the floppy is properly inserted and the floppy door latch is closed.

6. Contents -- Development System Floppies

The following files are included in the 1.3 release XENIX Development System package.

floppy #1 :

/bin/adb	/bin/ar
/bin/as	/bin/as_1.1
/bin/cb	/bin/cc
/bin/gets	/bin/hdr
/bin/lorder	/bin/make
/bin/nm	/bin/ranlib
/bin/regcmp	/bin/size
/bin/strings	/bin/strip
/bin/time	/bin/tsort
/lib/Mbcrt0.o	/lib/Mbmcrt0.o

floppy #2 :

/lib/Mblibcfp.a	/lib/Mbseg.o
/lib/Mblibdbm.a	/lib/Mblibtermcap.a
/lib/Mblibtermlib.a	/lib/Mblibl.a
linked to /lib/Mblibtermcap.a	/lib/Mbliбln.a
/lib/Mbliby.a	linked to /lib/Mblibl.a
/lib/Mblibc.a	/lib/Mblibx.a
/lib/Mblibm.a	/lib/Mblibcurses.a
/lib/Sblibc.a	/lib/Sblibx.a
/lib/Sblibcfp.a	/lib/Sblibcurses.a
/lib/Sblibdbm.a	/lib/Sblibl.a
/lib/Sbliбln.a	/lib/Sblibtermcap.a
linked to /lib/Sblibl.a	/lib/Sblibtermlib.a
/lib/Sbliby.a	linked to /lib/Sblibtermcap.a
/lib/Sbcrt0.o	/lib/Sbmcrt0.o
/lib/Sbseg.o	/lib/Sbsegimp.o

floppy #3 :

/lib/Sblibm.a	/lib/cpp
/lib/p0	/lib/p1
/lib/p2	/lib/p3
/usr/bin/admin	/usr/bin/cdc
/usr/bin/rmdel	/usr/bin/cref
linked to /usr/bin/cdc	/usr/bin/help

floppy #4 :

/usr/bin/comb	/usr/bin/ctags
/usr/bin/delta	/usr/bin/get
/usr/bin/lex	/usr/bin/lint
/usr/bin/m4	/usr/bin/mkstr
/usr/bin/prof	/usr/bin/prs
/usr/bin/ratfor	/usr/bin/sact
/usr/bin/unget	/usr/bin/sccsdiff
linked to /usr/bin/sact	/usr/bin/spline
/usr/bin/val	/usr/bin/xref
/usr/bin/xstr	/usr/bin/yacc
/usr/include/a.out.h	/usr/include/sys/a.out.h
/usr/include/ar.h	linked to /usr/include/a.out.h
/usr/include/assert.h	/usr/include/sys/assert.h
/usr/include/core.h	linked to /usr/include/assert.h
/usr/include/ctype.h	/usr/include/curses.h
/usr/include/dbm.h	/usr/include/dumprestor.h
/usr/include/errno.h	/usr/include/execargs.h
/usr/include/fcntl.h	/usr/include/grp.h
/usr/include/math.h	/usr/include/mnttab.h
/usr/include/pwd.h	/usr/include/regexp.h
/usr/include/setjmp.h	

floppy #5 :

/usr/include/sgtty.h	/usr/include/signal.h
/usr/include/stand.h	/usr/include/stdio.h
/usr/include/string.h	/usr/include/sys/acct.h
/usr/include/sys/buf.h	/usr/include/sys/callo.h
/usr/include/sys/conf.h	/usr/include/sys/dir.h
/usr/include/sys/fblk.h	/usr/include/sys/file.h
/usr/include/sys/filsys.h	/usr/include/sys/ino.h
/usr/include/sys/inode.h	/usr/include/sys/iobuf.h
/usr/include/sys/ioctl.h	/usr/include/sys/lock.h

/usr/include/sys/locking.h
/usr/include/sys/mount.h
/usr/include/sys/proc.h
/usr/include/sys/relsym.h
/usr/include/sys/sd.h
/usr/include/sys/sites.h
/usr/include/sys/space.h
/usr/include/sys/sysinfo.h
/usr/include/sys/text.h
/usr/include/sys/times.h
/usr/include/sys/tty.h
/usr/include/sys/ulimit.h
/usr/include/sys/utsname.h
/usr/include/termio.h
/usr/include/ustat.h
/usr/include/varargs.h
/usr/lib/cref/atab
/usr/lib/cref/crpost
/usr/lib/cref/eign
/usr/lib/cref/upost
/usr/lib/help/bd
/usr/lib/help/cm
/usr/lib/help/co
/usr/lib/help/default
/usr/lib/help/he
/usr/lib/help/rc
/usr/lib/help/ut
/usr/lib/lint1
/usr/lib/llib-lc
/usr/lib/llib-lcurses
/usr/lib/llib-lm
/usr/lib/llib-port
/usr/lib/xrefa
/usr/lib/yaccpar
/once/init.soft

/usr/include/sys/map.h
/usr/include/sys/param.h
/usr/include/sys/reg.h
/usr/include/sys/relsym86.h
/usr/include/sd.h
 linked to /usr/include/sys/sd.h
/usr/include/sys/stat.h
/usr/include/sys/systm.h
/usr/include/sys/timeb.h
/usr/include/sys/ttold.h
/usr/include/sys/types.h
/usr/include/sys/user.h
/usr/include/sys/var.h
/usr/include/time.h
/usr/include/utmp.h
/usr/lib/cref/aign
/usr/lib/cref/cign
/usr/lib/cref/ctab
/usr/lib/cref/etab
/usr/lib/help/ad
/usr/lib/help/cb
/usr/lib/help/cmds
/usr/lib/help/de
/usr/lib/help/ge
/usr/lib/help/prs
/usr/lib/help/un
/usr/lib/lex/ncform
/usr/lib/lint2
/usr/lib/llib-lc.ln
/usr/lib/llib-lcurses.ln
/usr/lib/llib-lm.ln
/usr/lib/llib-port.ln
/usr/lib/xrefb
/etc/soft.perms

floppy #6 :

/usr/bin/dosld	/usr/include/dos/assert.h
/usr/include/dos/ctype.h	/usr/include/dos/errno.h
/usr/include/dos/fcntl.h	/usr/include/dos/math.h
/usr/include/dos/setjmp.h	/usr/include/dos/signal.h
/usr/include/dos/spawn.h	/usr/include/dos/stdio.h
/usr/include/dos/time.h	/usr/include/dos/sys/stat.h
/usr/include/dos/sys/timeb.h	/usr/include/dos/sys/types.h
/usr/include/dos/sys/utime.h	/usr/lib/dos/Sdcrt0.o
/usr/lib/dos/Sdlibc.a	/usr/lib/dos/Sdlibcfp.a
/usr/lib/dos/Mdlibc.a	/usr/lib/dos/Mdlibcfp.a
/usr/lib/dos/Mdlibm.a	/usr/lib/dos/Mdcrt0.o
/usr/lib/dos/Ldcrt0.o	/usr/lib/dos/Ldlibc.a

floppy #7 :

/usr/lib/dos/Ldlibcfp.a	/usr/lib/dos/Ldlibm.a
/usr/lib/dos/Sdlibm.a	/usr/lib/dos/rawmode.o
/once/dos.perms	/once/init.dos

Chapter 7

As_1.1: An Assembler

7.1 Introduction

This document describes the usage and input syntax of the XENIX 8086 assembler, *as_1.1*, an assembler that produces an output file containing relocation information and a complete symbol table. The output is acceptable to the XENIX loader *ld*(CP), which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

This chapter does not teach assembly language programming, nor does it give a detailed description of 8086 operation codes. For information on these topics, you will need other references.

7.2 Command Usage

As_1.1 is invoked as follows:

as_1.1 [-l] [-o *output*] *file*

If the optional **-l** argument is given, an assembly listing is produced which includes the source, the assembled (binary) code, and any assembly errors. This name of the listing file has an **.L** extension.

The output of the assembler is by default placed on the file *file.o* in the current directory; The **-o** flag causes the output to be placed on the named file.

7.3 Lexical Conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), constants, and operators.

7.3.1 Identifiers

Identifiers begin with a period, underscore, or letters. Identifiers may contain periods, underscores, or letters (case is significant), and digits. Only the first eight characters are significant.

7.3.2 Constants

A hex constant consists of a backslash character (/) followed by a sequence of digits and one of the letters "a", "b", "c", "d", "e", and "f" (any of which may be capitalized). The letters are interpreted with the following values:

HEX	DECIMAL
a	10
b	11
c	12
d	13
e	14
f	15

An octal constant consists of a series of digits, preceded by the tilde (~) character. The digits must be in the range 0-7.

A decimal constant consists simply of a sequence of digits. The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

7.3.3 Blanks

Blank and tab characters may be freely interspersed between tokens, but may not be used within tokens (except in character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

7.3.4 Comments

Comments are ignored by the assemler. The vertical bar (|) introduces comments, which extends to the end of the line on which it appears.

7.4 Segments

Assembled code and data fall into three segments: the *text* segment, the *data* segment, and the *bss* segment. The *text* segment is the one in which the assembly begins, and it is the one into which instructions are typically placed. The XENIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link editor *ld*(CP)(using its **-i** option) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for storing data or instructions that may be modified during execution. Anything that may be stored in the text segment may be put into the data segment. In programs with write-protected, shareable text segments, the data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment. If the text segment is pure, the data segment is in an address space of its own, starting at location zero(0).

The bss segment may not contain any explicitly initialized code or data. The

length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. For more information, see the discussion of the **bss** directive.

7.5 The Location Counter

The special symbol, "dot" (.), is the location counter. Its value at any time is the offset within the appropriate segment from the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change. Furthermore, the value of dot may not decrease. If the effect of the assignment is to increase the value of dot, then the required number of null bytes is generated.

7.6 Statements

A source program is composed of a sequence of statements. Statements are separated by newlines. There are four kinds of statements:

- Null statements
- Expression statements
- Assignment statements
- Keyword statements

The format for most 8086 assembly language source statements is:

[*labelfield*] op-code [*operandfield*] [*comment*]

Any kind of statement may be preceded by one or more labels.

7.6.1 Labels

There are two kinds of labels: name labels and numeric labels. A name label consists of an identifier followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

A numeric label consists of a string of digits 0 to 9 and a dollar-sign ($) followed by a colon (:). Such a label serves to define local symbols of the form

n $

where *n* is the digit of the label. The scope of the numeric label is the labeled block in which it appears. As an example, the label "9$" is defined only between the labels *label1* and *label2*:

```
label1:
9$:         .byte 0
            .
            .
            .
label2:     .word a
```

As in the case of name labels, a numeric label assigns the current value and type of dot to the symbol.

7.6.2 Null statements

A null statement is an empty statement (which may, however, have labels and a comment). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

7.6.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler places the value of the expression in the output, together with the appropriate relocation bits.

7.6.4 Assignment statements

An assignment statement consists of an identifier, an equal sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of an expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a nonlocally defined global symbol.

As mentioned, it is permissible to assign to the location counter. It is required, however, that the type of the expression assigned be of the same type as dot, and it is forbidden to decrease the value of dot. In practice, the most common assignment to dot has the form

$. = . + n$

for some number n; this has the effect of generating n null bytes.

7.6.5 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

7.7 Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, and operators. Each expression has a type.

Arithmetic is two's complement. All operators have equal precedence, and expressions are evaluated strictly left to right.

7.7.1 Expression operators

The operators are:

Operator	Description
(blank)	Same as +
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Logical OR
&&	Logical AND
!!	Logical NOT
>	Right Shift
<	Left Shift

7.7.2 Types

The assembler deals with expressions, each of which may be of a different *type*. Most types are attached to the keywords and are used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression.

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefinied external. If such a symbol is declared, the link editor *ld*(CP) must be used to load the assembler's output with another routine that defines teh undefined reference.

absolute

An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text

The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of dot is text 0.

data

The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run, since previously loaded programs may have data segments. After the first *.data* statement, the value of dot is data 0.

bss

The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of dot is bss 0.

external absolute, text, data, or bss

Symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

other types

Each keyword known to the assembler has a type that is used

to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

7.7.3 Type Propagation in Expressions

When operands are combined by expression operators, the result has a type that depends on the types of the operands and on the operator. The rules involved are complex, but are intended to be sensible and predictable. For purposes of expression evaluation the important types are:

undefined
absolute
text
data
bss
undefined external
other

The combination rules are as follows:

— If one of the operands is undefined, the result is undefined.

— If both operands are absolute, the result is absolute.

— If an absolute is combined with one of the other types mentioned above, the result has the other type.

— If two operands of other type are combined, the result has the numerically larger type.

— An other type combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

\- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others

It is illegal to apply these operators to any but absolute symbols.

7.8 Assembler Directives

As_1.1 supports a number of assembler directives (also called "Pseudo-operations" or "pseudo-ops"). The keywords listed below introduce statements that influence the later operations of the assembler. The metanotation

[*item*] ...

means that 0 or more instances of the given *item* may appear. Also, boldface tokens are literals, italic words are substitutable.

7.8.1 .even

If the location counter is odd, it is advanced by one so the next statement will be assembled at a word boundary. This is useful for forcing storage allocation to be on a word boundary after a **.byte** or **.ascii** directive.

7.8.2 .float, .double

.float *float*

The **.float** pseudo operation accepts as its operand an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, them an optional "e" or "E", followed by an optionally signed integer. The string is interpreted as a floating point number. The difference between **.float** and **.double** is in the number of bytes for the result. The **.float** sets aside four bytes, while **.double** sets aside eight bytes.

7.8.3 .globl

.globl *name* [, *name*] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor *ld*(CP) may be used to combine this routine with other routines that refer to these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others

which define the symbols. It is possible to force the assembler to make all otherwise undefined symbols external.

7.8.4 .text, .data, .bss

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment, respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and dot moved about by assignment.

7.8.5 .comm

The format of the **.comm** statement is

.comm *name*

Provided the *name* is not defined elsewhere, this statement is equivalent to **.globl**. That is, the type of *name* is "undefined external", and its size is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols that are not otherwise defined, and that have a nonzero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

7.8.6 .insrt

The format of a **.insrt** is:

.insrt "*filename*"

where *filename* is any valid XENIX filename. Note that the *filename* must be enclosed within double quotation marks.

The assembler attempts to open this file for input. If it succeeds, source lines are read from it until the end of file is reached. If *as* is unable to open the file, the following error message is printed:

Cannot open insert file

This statement is useful for including a standard set of comments or symbol assignments at the beginning of a program. It is also useful for breaking up a large source program into easily manageable pieces.

The maximum depth of nexted insertions with **.insrt** is ten.

System call names are not predefined. They may be found in the file */usr/include/sys.s*.

7.8.7 .ascii, .asciz

The **.ascii** directive translates character strings into their 7-bit ASCII (represented as 8-bit bytes) equivalents for use in the source program. The format of the **.ascii** directive is as follows:

.ascii */string/*

where *string* contains any character valid in a character constant. Obviously, a *newline* must not appear within the character string. (It can be represented by the escape sequence "\n"). Note that the slash is the delimter character. This may be any be any character not appearing in *character-string*.

Several examples follow:

Hex Code Generated	Statement:
22 68 65 6C 6C 6F 20 74 68 65 72 65 22	.ascii /"hello there"/
77 61 72 6E 69 6E 67 20 2D 07 07 20 0A	.ascii "Warning-\007\007 \n"
61 62 63 64 65 66 67	.ascii *abcdefg*

The **.asciz** directive is equivalent to the **.ascii** directive with a zero (null) byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string. Null terminated strings are used as arguments to some XENIX system calls.

7.8.8 .list, .nlist

These pseudo-directives control the assembler output listing. These, in effect, temporarily override the **-l** switch to the assembler. This is useful when certain portions of the assembly output is not necessarily desired on a printed listing.

.list *Turns the listing on*
.nlist *Turns the listing off*

7.8.9 .blkb, .blkw

The **.blkb** and **.blkw** directives are used to reserve blocks of storage: **.blkb** reserves bytes, **.blkw** reserves words.

The format is:

```
.blkb   [expression]
.blkw   [expression]
```

where *expression* is the number of bytes or words to reserve. If no argument is given a value of 1 is assumed. The expression must be absolute, and defined during pass 1.

This is equivalent to the statement

```
. = .+ expression
```

but has a much more transparent meaning.

7.8.10 .byte, .word

The **.byte** and **.word** directives are used to reserve bytes and words and to initialize them with certain values.

The format is:

```
.byte   [expression]
.word   [expression]
```

The **.byte** directive reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression.

The semantics for **.word** are identical, except that 16-bit words are reserved and initialized.

7.8.11 .end

The **.end** directive indicates the physical end of the source program. The format is:

```
.end    [expression]
```

where *expression* is an optional argument which, if present, indicates the entry point of the program, i.e., the starting point for execution. If the entry point of a program is not specified during assembly, it defaults to zero.

Every source program must be terminated with a .end statement. Inserted files that contain a .end statement will terminate assembly of the entire program, not just the inserted portion.

7.9 Machine Instructions

The 8086 instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. In the table that follows, with some exceptions, an instruction that operates on a byte operand will have a "b" suffix on the opcode.

The 8086 instruction mnemonics which follow are implemented by the Microsoft 8086 assembler described in this section. Some of the opcodes are not found in any other 8086 manual; for example, the some of the branch instructions are specific to this assembler. These branch instructions expand into a jump on the inverse of the condition specified, followed by an an unconditional intra-segment jump. The operand field format for the branch opcodes is the same as the operand field for the jump long opcodes. The opcodes that are implemented only in this assembler are annotated by an asterisk, and are fully defined and described in this document.

8086 Assembler Opcodes

Opcode	Description
aaa	ascii adjust for addition
aad	ascii adjust for division
aam	ascii adjust for multiply
aas	ascii adjust for subtraction
adc	add with carry
adcb	add with carry
add	add
addb	add
and	logical AND
andb	logical AND
*beq	long branch equal
*bge	long branch grt or equal
*bgt	long branch grt
*bhi	long branch on high
*bhis	long branch high or same
*ble	long branch les or equal
*blo	long branch on low
*blos	long branch low or same
*blt	long branch less than
*bne	long branch not equal
*br	long branch
call	intra segment call

calli	inter segment call
cbw	convert byte to word
clc	clear carry flag
cld	clear direction flag
cli	clear interrupt flag
cmc	complement carry flag
cmp	compare
cmpb	compare
cmps	compare string
cmpsb	compare string
cwd	covert word to double word
daa	decimal adjust for addition
das	decimal adjust for subtraction
dec	decrement by one
decb	decrement by one
div	division unsigned
divb	division unsigned
hlt	halt
idiv	integer division
idivb	integer division
imul	integer multiplication
imulb	integer multiplication
in	input byte
inc	increment by one
incb	increment by one
int	interrupt
into	interrupt if overflow
inw	input word
iret	interrupt return
j	short jump
ja	short jump if above
jae	short jump if above or equal
jb	short jump if below
jbe	short jump if below or equal
jcxz	short jump if CX is zero
je	short jump on equal
jg	short jump on greater than
jge	short jump greater than or equal
jl	short jump on less than
jle	short jump on less than or equal
jmp	jump
jmpi	inter segment jump
jna	short jump not above
jnae	short jump not above or equal
jnb	short jump not below
jnbe	short jump not below or equal
jne	short jump not equal
jng	short jump not greater
jnge	short jump not greater or equal

jnl	short jump not less
jnle	short jump not less or equal
jno	short jump not overflow
jnp	short jump not parity
jns	short jump not sign
jnz	short jump not zero
jo	short jump on overflow
jp	short jump if parity
jpe	short jump if parity even
jpo	short jump if parity odd
js	short jump if signed
jz	short jump if zero
lahf	load AH from flags
lds	load pointer using DS
lea	load effective address
les	load pointer using ES
lock	lock bus
lodb	load string byte
lodw	load string word
loop	loop short label
loope	loop if equal
loopne	loop if not equal
loopnz	loop is not zero
loopz	loop if zero
mov	move
movb	move byte
movs	move string
movsb	move string byte
mul	multiplication unsigned
mulb	multiplication unsigned
neg	negate
negb	negate
nop	no op
not	logical NOT
notb	logical NOT
or	logical OR
orb	logical OR
out	output byte
outw	output word
pop	pop from stack
popf	pop flag from stack
push	push onto stack
pushf	push flags onto stack
rcl	rotate left through carry
rclb	rotate left through carry
rcr	rotate right through carry
rcrb	rotate right through carry
rep	repeat string operation
repnz	repeat string operation not zero

repz	repeat string operation while zero
ret	return from procedure
reti	return from intersegment procedure
rol	rotate left
rolb	rotate left
ror	rotate right
rorb	rotate right
sahf	store AH into flags
sal	shift arithmetic left
salb	shift arithmetic left
sar	shift arithmetic right
sarb	shift arithmetic right
sbb	subtract with borrow
sbbb	subtract with borrow
scab	scan string
shl	shift logical left
shlb	shift logical left
shr	shidr logical right
shrb	shidr logical right
stc	set carry flag
std	set direction flag
sti	set interrupt enable flag
stob	store byte string
stow	store word string
sub	subtraction
subb	subtraction
test	test
testb	test
wait	wait while TEST pin
xchg	exchange
xchgb	exchange
xlat	translate
xor	xclusive OR
xorb	xclusive OR

7.10 Addressing Modes

The 8086 assembler provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory, or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways.

7.10.1 Register Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all the instruction forms. This is because the register

"addresses" are encoded in the instructions with just a few bits, and because these operations are performed entirely within the CPU. Registers may serve as source operands, destination operands, or both. Examples of register addressing follow:

```
sub     cx,di
mv      ax,/3*4
mv      /3*4/,ax
mov     ax,*1
```

7.10.2 Immediate Operands

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue. It is possible that no bus cycles will be needed to obtain an immediate operand. An immediate operand is always a constant value and can only be used as a source operand.

The assembler can accept both 8 and 16 bit operands. It does not perform any checking on the operand size, but determines the size of the operand by the following symbols:

```
*expr    an 8 bit immediate
#expr    a 16 bit immediate
```

Examples of immediate addressing follow:

```
mov     cx,*PAGSIZ/2
mov     cx,#PAGSIZ/2
mov     map,#PAGSIZ/2
mov     map,*PAGSIZ/2
```

7.11 Memory Addressing Modes

When reading or writing a memory operand, a value called the offset is required. This offset value, also called the *effective address* is the operand's distance in bytes from the beginning of the segment in which it resides.

7.11.1 Direct Addressing

Direct addressing is the simplest memory addressing mode since no registers are involved. The effective address is taken directly from the displacement field of the instruction. It is typically used to access simple (scalar) variables. Examples

of direct addressing follow:

```
push    *6(bp)
mov     cx,#256
add     si,*4
```

7.11.2 Register Indirect Addressing

The effective address of a memory operand may be taken from a base or index register. One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. Indirect addressing is denoted by an ampersand (\@) preceding the operand. Examples of indirect addressing follow:

```
popl    rr0,@r15
calli   @moncall
```

7.11.3 Based Addressing

In based addressing, the effective address is the sum of a displacement value and the content of register **bx** or **bp**. Based addressing also provides a straightforward way to address structures which may be located in different places in memory. A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register. An example of based addressing follows:

```
mov     *2(si),#/1000
```

7.11.4 Indexed Addressing

In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register. Indexed addressing often is used to access elements in an array. The displacement locates the beginning of the array, and the value of the index register selects one element. Since all array elements are the same length, simple arithmetic on the index register will select any element. An example of indexed addressing follows:

```
mov     #_cat,(bx)
```

7.11.5 Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement. Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack. Register **bp** can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements. Examples of based indexed addressing follow:

```
mov     (bx)(dx),_sym
mov     *2(bx)(dx),_sym
mov     #2(bx)(dx),_sym
```

7.12 Diagnostics

When syntactic errors occur, the line number and the file in which they occur is displayed. Errors in pass 1 cause cancellation of pass 2.

```
***ERROR*** syntax error, line nnn
file: eee errors
```

where *nnn* represents the line number(s) in error, and *eee* represents the total number of errors.

Name

as_1.1 – assembler

Synopsis

as_1.1 [–] [– **o** objfile] file ...

Description

As_1.1 assembles the concatenation of the named files. If the optional first argument – is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file *objfile*. The output format is INTEL 8086 Relocatable Format. See the manual page 86rel(F) for a description of the output format.

Files

/bin/as_1.1	this assembler
file.o	object

See Also

ld(CP), nm(CP), adb(CP), a.out(F), 86rel(F)
XENIX *Programmer's Reference*

Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
]	Parentheses error
<	String not terminated properly
*	Indirection used illegally

a	Error in address
b	Branch instruction is odd or too remote
e	Error in expression
f	Error in local ('f' or 'b') type symbol
g	Garbage (unknown) character
i	End of file inside an if
m	Multiply defined symbol as label
o	Word quantity assembled at odd address
p	'.' different in pass 1 and 2
r	Relocation error
u	Undefined symbol
x	Syntax error

Notes

Syntax errors can cause incorrect line numbers in following diagnostics.

Name

sdget, sdfree – Attachs and detachs a shared data segment.

Syntax

```
#include <sd.h>

char *sdget(path, flags, [size, mode])
char *path;
int flags, mode;
int size;

int sdfree(addr);
char *addr;
```

Description

Sdget attachs a shared data segment to the data space of the current process. The actions performed are controlled by the value of *flags*. *Flags* values are constructed by OR-ing flags from the following list:

SD_RDONLY
: Attach the segment for reading only.

SD_WRITE
: Attach the segment for both reading and writing.

SD_CREAT
: If the segment named by *path* exists and is not in use (active), this flag will have the same effect as creating a segment from scratch. Otherwise, the segment is created according to the values of *size* and *mode*. Read and write access to the segment is granted to other processes based on the permissions passed in *mode*, and functions the same as those for regular files. Execute permission is meaningless. The segment is initialized to contain all zeroes.

SD_UNLOCK
: If the segment is created because of this call, the segment will be made so that more than one process can be between sdenter and sdleave calls.

Sdfree detachs the current process from the shared data segment that is attached at the specified address. If the current process has done an *sdenter* but not a *sdleave* for the specified segment, an *sdleave* will be done before detaching the segment.

When no process remains attached to the segment, the contents of that segment disappear, and no process can attach to the segment without creating it by using the SD_CREAT flag in *sdget*. *Errno* is set to EEXIST if a process tries to create a shared data segment that exists and is in use. *Errno* is set to ENOTNAM if a process attempts an *sdget* on a file that exists but is not a shared data type.

Notes

Use of the SD_UNLOCK flag on systems without hardware support for shared data may cause severe performance degradation.

It is recommended that *sdget* and other shared data functions be reserved for large model programs only. Small or middle model programs that attempt to use shared data may run out of available memory.

Sdget automatically increments the process's original break value to the memory location immediately after the shared data segment. This affects subsequent *sbrk* or *brk* calls which attempt to restore the original break value. In particular, attempts to restore the break value to its value before the *sdget* call will cause an error.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine may be linked using the linker option **-lx**.

Return Value

On successful completion, the address at which the segment was attached is returned. Otherwise, -1 is returned, and *errno* is set to indicate the error. *Errno* is set to EINVAL if a process does an *sdget* on a shared data segment to which it is already attached. *Errno* is set to EEXIST if a process tries to create a

shared data segment that exists and is in use. *Errno* is set to ENOTNAM if a process attempts an *sdget* on a file that exists but is not a shared data type.

See Also

sdenter(S), sdgetv(S), sbrk(S)