ALTOS

RELEASE NOTES

ACS-586 XENIX C Compiler (x.out Format)
Version 1.1a
May 29, 1984

Introduction

These diskettes contain the XENIX x.out C compiler, also referred to as the "medium model" C compiler. Included with this release is the C compiler, assembler, linker, and other programs that are required to support x.out programs.

Under the a.out model, C programs were limited to 64K bytes of code space and 64K bytes of data/stack space. This is referred to as "small model" and is the only model supported by the a.out C compiler. The compiler supplied with this release supports either small or "medium" model, which means that a maximum of 192K bytes of code and 64K bytes of data space are supported.

All of the utilities required to support code generation for the x.out format are supplied in this release. Several deficiencies of the previous x.out release have been rectified in this version. All known compiler optimizer bugs have been corrected. The linker has been enhanced to support link editing of large programs. In addition, the libraries for the lex(1) utility and the C preprocessor, /lib/cpp, are provided on this release.

SOFTWARE

Changes from Version 1.0a

This release of the x.out C compiler is different from the previous release in the following ways:

1. The -S switch to the previous x.out linker caused it to emit a symbol table file, to be integrated later with the binary by the "nlcv" program. The linker in this release does this step automatically, so the -S switch is no longer necessary. The new linker will accept -S in its command line, but will not perform any action. The nlcv program supplied with previous releases is no longer necessary, and is not supplied with this release. Current linker command lines (in makefiles or shell scripts) should be changed to eliminate any calls to nlcv.

2. The -C switch in the previous release requested the linker to consider the case of symbol name as significant. This option now means to disregard the case of symbol names when considering uniqueness.

3. In the previous x.out linker release, medium model programs were linked with "seg.o" for both small and medium model. This implementation requires the use of "Mseg.o" for medium model. Seg.o is still used for small model.

Changes from a.out

When switching from the a.out to x.out C compiler, the following notes should be kept in mind:

1. The -M option has been added to cc(1) and as(1) to support compilation of medium model programs. If the -M option is not used, then user programs are compiled small model. There may be a small performance advantage to compiling a program small model versus medium model. This is due to the fact that in some cases long forms of instructions are used in the medium model case. These instructions take fractionally longer to execute than their small model counter-parts. A program compiled medium model will be larger than its small model version for the same reason.

2. The options to ld(1) have changed from a.out. When linking to a library using the "-l" option, the library name must be a fully qualified path name. In addition, all libraries must be run through ranlib(1) before linking.

3. Intermediate object files (".o" files) in a.out format can not be linked with x.out object files. All ".o" files must be re-compiled and all libraries re-built in the x.out format before the linker can produce an executable binary.

4. The libraries contained in this release are prefaced by "M" or "S" , corresponding to medium or small model. The appropriate libraries will be linked automatically by cc(1). For this reason, it is recommended that cc(1) be used to link programs.

5. All executable binaries that have been compiled small model, whether in the a.out or x.out formats, are expected to be compatible with the Intel 286 processor. However, medium model binaries may not be compatible with future releases of XENIX for the 286. All medium model programs may have to be re-compiled when the time comes to port them to the 286.

6. A new /usr/include/stdio.h has been included with this release. This stdio.h will not work properly with the small model (a.out) C Compiler and vice versa.

Known Bugs

The following bugs are known to exist in this release of the x.out C compiler;

1. There is a bug in /lib/cl that prevents the growth of the stack by more than 32K bytes at a time. This means that locally defined data arrays can not be larger than 32767 bytes. To work around this problem simply declare the data structure outside of a procedure; i.e. use a global definition.

2. The medium model linker (/bin/ld) does not currently support the "-e" or "-r" flags.

3. Some uses of the -= construction do not compile correct code when the type of the assignment is a float or double. Such a construction is:

```
float a = 0.0;

a -= 1.0 - .5;/* delivers wrong result */
```

The use of the equivalent statement a = a - (1.0 - 0.5) is recomended. This problem occurs without respect to model or optimization.

Installation Instructions

Note: the installation procedure for the x.out C compiler will over-write some a.out C compiler files that may exist on your system.

To install either the x.out or a.out C compiler on your system, insert the proper floppy disk into the floppy drive, login as root, and enter the following commands;

```
cd /
umask 0
tar xv
```

These commands will install all the programs and libraries in their correct places with the correct permissions. Switching between C compilers is not recommended but may be accomplished by inserting the a.out (or x.out) C compiler floppy disk into the floppy drive and following the steps above.

Name

cc - Invokes the C compiler.

Syntax

cc [options] filename ...

Description

Cc is the XENIX C compiler command. It creates executable programs by compiling and linking the files named by the filename arguments. Cc copies the resulting program to the file a.out.

The filename can name any C or assembly language source file or any object or library file. C source files must have a ``.c'' filename extension. Assembly-language source files must ``.s'', object files ``.o'', and library files ``.a'' extensions. Cc invokes the C compiler for each C source file and copies the result to an object file whose basename is the same as the source file but whose extension is ``.o''. Cc invokes the XENIX assembler, as , for each assembly source file and copies the result to an object file with extension ``.o''. Cc ignores object and library files until all source files have been compiled or assembled. It then invokes the XENIX link editor, ld , and combines all the object files it has created together with object files and libraries given in the command line to form a single program.

Files are processed in the order they are encountered in the command line, so the order of files is important. Library files are examined only if functions referenced in previous files have not yet been defined. Library files must be in ranlib(1) format, that is, the first member must be named __.SYMDEF, which is a dictionary for the library. The library is searched repeatedly to satisfy as many references as possible. Only those functions that define unresolved references are concatenated. A number of ``standard'' libraries are searched automatically. These libraries support the standard C library functions and program startup routines. Which libraries are used depends on the program's memory model (see ``Memory Models'' below). The entry point of the resulting program is set to the beginning of the ``main'' program function.

There are the following options:

-P

Preprocesses each source file and copies the result to a file whose basename is the same as the source but whose extension is ``.i''. Preprocessing performs the actions specified by the preprocessing directives.

-E Preprocesses each source file as described for -P , but copies the result to the standard output. The option also places a #line directive with the current input line number and source file name at the beginning of output for each file.

-C Preserves comments when preprocessing a file with -E or -P. That is, comments are not removed from the preprocessed source. This option may only be used in conjunction with -E or -P .

-D*name* [= *string*]
Defines *name* to the preprocessor as if defined by #define in each source file. The form ``-D*name*'' sets *name* to 1. The form ``-D*name* = *string*'' sets *name* to the given *string*.

-I*pathname*
Adds *pathname* to the list of directories to be searched when an #include file is not found in the directory containing the current source file or whenever angle brackets (< >) enclose the filename. If the file cannot be found in directories in this list, directories in a standard list are searched.

-X Removes the standard directories from the list of directories to be searched for #include files.

-v2|3
sets a flag which provides for XENIX 2.x compatibility (-v2) or XENIX 3.0 (-v3). The default is -v2.

-i Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and may be shared by all users executing the file. The option is implied when creating middle model programs.

-K*num*
Removes stack probes from a program. Stack probes are used to detect stack overflow on entry to program routines.

-F *num*
Tells the linker to use a fixed stack size of "num" hex bytes. The argument must be supplied as "0xNNNN".

-M The generated program is to be medium model.

-c Creates a linkable object file for each source file but does not link these files. No executable program is

created.

-o filename
Defines filename to be the name of the final executable program. This option overrides the default name a.out.

-llibrary
Searches library for unresolved references to functions. The library must be an object file archive library in ranlib format.

-O Invokes the object code optimizer.

-S Creates an assembly source listing of the compiled C source file and copies this listing to the file whose basename is the same as the source but whose extension is ``.s''. This file is suitable for assembly using as(1).

-L Creates an assembler listing file containing assembled code and assembly source instructions. The listing is copied to the file whose basename is the same as the source but whose extension is ``.L''. This options suppresses the ``-S'' option.

Many options (or equivalent forms of these options) are passed to the link editor as the last phase of compilation. The -M option is passed to specify program model. The -i and -p are passed to specify other characteristics of the final program.

The -D and -I options may be used several times on the command line. The -D option must not define the same name twice. These options affect subsequent source files only.

Memory Models
Cc can create programs for two different memory models: small and middle. In addition, small model programs can be pure or impure.

Impure-Text Small Model
These programs occupy one 64 Kbyte physical segment in which both text and data are combined. Cc creates impure small model programs by default. They can also be created using the ``-Ms'' option.

Pure-Text Small Model
These programs occupy two 64 Kbyte physical segments. Text and data are in separate segments. The text is read-only and may be shared by several processes at once. The maximum program size is 128 Kbytes. Pure small model programs are created using the ``-i''

option.

Middle (or Medium) Model
These programs occupy several physical segments, but only one segment contains data. Text is divided among as many segments as required. Special call and returns are used to access functions in other segments. Text can be any size, up to 192K. Data must not exceed 64 Kbytes. Middle models programs are created using the ``-M'' option. These programs are always pure.

Small and middle model object files can only be linked with object and library files of the same model. It is not possible to combine small and medium model object files in one executable program. Cc automatically selects the correct small and middle versions of the standard libraries based on the -M option. It is up to the user to make sure that all of his own object files and private libraries are properly compiled in the appropriate model.

The special calls and returns used in middle model programs may affect execution time. In particular, the execution time of a program which makes heavy use of functions and function pointers may differ noticably from small model programs.

In middle model programs, function pointers are 32 bits long. Programs making use of such pointers must be written carefully to avoid incorrect declaration and use of these variables. Lint(1) will help to check for correct use.

Files

/bin/cc

See Also

as(1), ar(1), ld(1), lint(1), ranlib(1)

Notes

Error messages are produced by the program that detects the error. These messages are usually produced by the C compiler, but may occasionally be produced by the assembler or the link loader.

All object module libraries must have an up to date ranlib directory.

Name

ld - Invokes the link editor.

Syntax

ld [options] filename...

Description

Ld is the XENIX link editor. It creates an executable program by combining one or more object files and copying the executable result to the file a.out. The filename must name an object or library file. These names must have the ``.o'' (for object) or ``.a'' (for archive library) extensions. If more than one name is given, the names must be separated by one or more spaces. If errors occur while linking, ld displays an error message; the resulting a.out file is unexecutable.

Ld concatenates the contents of the given object files in the order given in the command line. Library files in the command line are examined only if there are unresolved external references encountered from previous object files. Library files must be in ranlib(1) format, that is, the first member must be named __.SYMDEF, which is a dictionary for the library. The library is searched iteratively to satisfy as many references as possible and only those routines that define unresolved external references are concatenated. Object and library files are processed at the point they are encountered in the argument list, so the order of files in the command line is important. In general, all object files should be given before library files. Ld sets the entry point of the resulting program to the beginning of the first routine.

There are the following options:

-F num
: Sets the size of the program stack to hex num bytes. The stack size must be supplied.

-i
: Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and shared by all users executing the file.

-Ms
: Creates small model program and checks for error, such as fixup overflow. This option is reserved for object files compiled or assembled using the small model configuration. This is the default model if no -M option is given.

-Mm
: Creates middle model program and checks for errors.

This option is reserved for object files compiled or assembled using the middle model configuration. -M with no model specifed defaults to medium. These options imply -i .

-o name
Sets the executable program filename to name instead of a.out.

-l library name
Searches the named library to supply archived routines.

-m Supply a link map.

-n length
Truncates symbol name sizes to "length".

-v 2|3
Sets compatibility mode; 2 is XENIX 2.x compatibility, 3 is XENIX 3.0 compatibility.

-C Case of symbol names is not significant.

-s Strip the symbol table from the output.

-u symbol
Enters the "symbol" as undefined.

-S Does nothing. Retained for compatibility with previous linker.

Ld should be invoked using the cc(1) instead of invoking it directly. Cc invokes ld as the last step of compilation, providing all the necessary C-language support routines. Invoking ld directly is not recommended since failure to give command line arguments in the correct order can result in errors.

Files

/bin/ld,	linker program
/lib/[MS]crt0.o,	C runtime binary
/lib/[MS]libc.a,	C library archive
/lib/[MS]libfp.a	C floating-point archive

See Also

as(1), ar(1), cc(1), ranlib(1)

Notes

The user must make sure that the most recent library versions have been processed with ranlib(1) before linking. If this is not done, ld cannot create executable programs using

these libraries.

NAME

a.out - assembler and link editor output

SYNOPSIS

```
#include <a.out.h>
#include <sys/relsym.h>
#include <sys/relsym86.h>
```

DESCRIPTION

A.out is the output file of the assembler as(1) and the link editor ld(1). Both programs make a.out executable if there were no errors and no unresolved external references.

A.out is the name given to all object files by default. In this context, when a file is referred to as being in x.out format, it contains an x.out header and may contain any of several symbol table and relocation record formats.

The a.out.h include file contains the header and extended header declarations and defined values for the header fields. It also contains declarations for the portable symbol structures used by library routines to access symbol tables.

The sys/relsym.h include file contains declarations and defined values for the symbol table and relocation structures used in an x.out file. The sys/relsym86.h include file contains declarations and descriptions of the relocation and symbol formats used in 8086 object files.

An x.out object file has seven sections: a header, an extended header, text, data, symbol table, text relocation records, and data relocation records (in that order). The symbols and relocation records may be empty if the program was loaded with the appropriate options to ld(1) or if they have been removed by strip(1).

When an x.out file is loaded into core for execution, three logical segments are set up: the text segment, the data segment, and stack segment. The data segment contains intialized data followed by bss, or blank storage space; bss is initialized to all 0. The text segment begins at the text relocation base (not necessarily 0) in the core image; the header itself is not loaded. If the text segment is to be impure (not write-protected), the data segment is immediately contiguous with the text segment.

If the text segment is pure, the data segment begins at the first page boundary following the text segment, and the text segment is not writeable by the program; if other processes are executing the same file, they will share the text segment. On some machines, the text and data segment locations

may be reversed.

If the text and data segments are separate (separate I & D), the text and data segments may begin at unrelated locations.

The stack segment will be located by the Xenix kernel, and on non-fixed stack machines is extended automatically as required. The data segment is only extended as requested by brk(2).

HEADER

In the header, the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

Layout information as given in the include file is:

```
struct xexec {                          /* x.out header */
        unsigned short x_magic;         /* magic number */
        unsigned short x_ext;           /* extended header size */
        long           x_text;          /* text size */
        long           x_data;          /* data size */
        long           x_bss;           /* bss size */
        long           x_syms;          /* symbol table size */
        long           x_reloc;         /* relocation size */
        long           x_entry;         /* entry point */
        char           x_cpu;           /* cpu type */
        char           x_relsym;        /* reloc & symbol format */
        unsigned short x_renv;          /* run-time environment */
};
```

Definition for the x.out magic number. The presence of this value in the first two bytes of a file indicates the file is in x.out format.

```
#define X_MAGIC     0x0206
```

Definitions for x_cpu. The first two defined bits are set if the bytes or words in the x.out header, symbol table and relocation records are not stored in the same order as on a PDP-11. The second group indicates the cpu for which the file was compiled or assembled.

```
#define XC_BSWAP   0x80     /* bytes swapped */
#define XC_WSWAP   0x40     /* words swapped */

#define XC_NONE    0x00     /* none */
#define XC_PDP11   0x01     /* PDP-11 */
#define XC_23      0x02     /* 23fixed PDP-11 */
#define XC_Z8K     0x03     /* Z8000 */
#define XC_8086    0x04     /* 8086 */
```

```
#define XC_68K      0x05    /* M68000 */
#define XC_Z80      0x06    /* Z80 */
#define XC_VAX      0x07    /* VAX 780/750 */
#define XC_16032    0x08    /* NS16032 */
#define XC_CPU      0x3f    /* cpu mask */
```

Definitions for x_relsym. The first group indicates the type of relocation attached, the second indicates the type of symbol table.

```
#define XR_RXOUT     0x00    /* x.out long form */
#define XR_RXEXEC    0x10    /* x.out short form */
#define XR_RBOUT     0x20    /* b.out format */
#define XR_RAOUT     0x30    /* a.out format */
#define XR_R86REL    0x40    /* 8086 relocatable */
#define XR_R86ABS    0x50    /* 8086 absolute */
#define XR_REL       0xf0    /* relocation mask */

#define XR_SXOUT     0x00    /* struct sym */
#define XR_SBOUT     0x01    /* struct bsym */
#define XR_SAOUT     0x02    /* struct asym */
#define XR_S86REL    0x03    /* 8086 relocatable */
#define XR_S86ABS    0x04    /* 8086 absolute */
#define XR_SUCBVAX   0x05    /* string table */
#define XR_SYM       0x0f    /* symbol mask */
```

Definitions for x_renv, the run-time environment. The first group indicates the version of Xenix for which the file was compiled. The XE_LTEXT and/or XE_LDATA bits are set if text and/or data addresses require more than sixteen bits. Other bits are set to describe the configuration of the text and data segments, to indicate if the fixed stack size field in the extended header is valid, and to indicate whether it is executable or linkable.

```
#define XE_V2       0x4000    /* up to version 2.3 */
#define XE_V3       0x8000    /* after version 2.3 */
#define XE_VERS     0xc000    /* version mask */

#define XE_RES      0x0080    /* reserved */
#define XE_LTEXT    0x0040    /* large model text */
#define XE_LDATA    0x0020    /* large model data */
#define XE_OVER     0x0010    /* text overlay */
#define XE_FS       0x0008    /* fixed stack */
#define XE_PURE     0x0004    /* pure text */
#define XE_SEP      0x0002    /* separate I & D */
#define XE_EXEC     0x0001    /* executable */
```

EXTENDED HEADER

The first two fields contain the sizes of text and data relocation attached to the file. The next two contain the base addresses for which text and data have already been

relocated. The fifth field is used on fixed stack machines to indicate the size of the stack segment required for execution.

```
struct  xext {
        long    xe_trsize;      /* text rel size */
        long    xe_drsize;      /* data rel size */
        long    xe_tbase;       /* text base */
        long    xe_dbase;       /* data base */
        long    xe_stksize;     /* stack size */
};
```

SYMBOL TABLE

The standard x.out symbol table (XR_SXOUT) is discussed here. For declarations of other supported symbol types, see the sys/relsym.h include file. Some flag values are only used internally by utility programs, and will not be found in object files.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the link editor ld(1) as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

Each symbol in the table has the structure given below, followed immediately by its name in the form of a null terminated string. The length of the symbol name, including the null terminator, must be no greater than SYMLENGTH. No effort is made to word align subsequent structures in the symbol table.

The first field encodes the symbol type, the last contains the value (usually the core address) of the symbol. The structure has been padded to allow portable use.

```
struct sym {
        unsigned short  s_type;
        unsigned short  s_pad;
        long            s_value;
};
```

```
#define SYMLENGTH     50
```

Definitions for s_type:

```
#define S_UNDEF    0x0000     /* undefined */
#define S_ABS      0x0001     /* absolute */
#define S_TEXT     0x0002     /* text */
#define S_DATA     0x0003     /* data */
#define S_BSS      0x0004     /* bss */
#define S_COMM     0x0005     /* internal */
#define S_REG      0x0006     /* register */
#define S_COMB     0x0007     /* internal */
#define S_TYPE     0x001f     /* mask */

#define S_FN       0x001f     /* file name */
#define S_EXTERN   0x0020     /* external bit */
```

The nlist structure, provided for compatibility with nlist(3).

```
struct  nlist {
        char        n_name[8];  /* symbol name */
        int         n_type;     /* type flag */
        unsigned    n_value;    /* value */
};
```

RELOCATION

If relocation information is present, it takes one of two forms. For linkable object files, the long form (XR_RXOUT) is used to enable references between modules to be resolved when linked together.

The first field encodes the segment referenced, the size (number of bytes) to relocate, and whether the relocation is relative. The second contains the symbol id, an index into the symbol table. The first symbol in the symbol table is referenced with 0. The symbol id is used to obtain the symbol value in order to perform external relocation. The last field contains the position within the current segment at which relocation is to be performed.

```
struct reloc {
        unsigned short  r_desc;     /* descriptor */
        unsigned short  r_symbol;   /* symbol id */
        long            r_pos;      /* position */
};
```

Definitions for r_desc. The first group encodes the segment referenced, the second the size of relocation, and the last whether relocation is relative.

```
#define RD_TEXT     0x0000
#define RD_DATA     0x4000
#define RD_BSS      0x8000
#define RD_EXT      0xc000
#define RD_SEG      0xc000

#define RD_BYTE     0x0000
#define RD_WORD     0x1000
#define RD_LONG     0x2000
#define RD_SIZE     0x3000

#define RD_DISP     0x0800
```

For executable files, the short form (XR_RXEXEC) is used to allow a single module to be relocated. This enables the link editor to relocate a file to run at the different text and data base addresses required for different machines, or to convert it to pure text, fixed stack, etc, without recompiling.

```
struct xreloc {
        long  xr_cmd;
};
```

The first bit is set if the relocation references the text segment, the second if the relocation involves four bytes (as opposed to two). The last field is the position, or offset, within the current segment at which the relocation is to be performed.

```
#define XR_CODE   0x80000000     /* code/text segment */
#define XR_LONG   0x40000000     /* long/short operand */
#define XR_OFFS   0x3fffffff     /* 30 bit offset */
```

SEE ALSO

as(1S), ld(1S), nm(1S), strip(1S), brk(2), nlist(3)