Split debug symbols for pkgsrc builds

Short report after Google Summer of Code 2016

Leonardo Taccari
leot@NetBSD.org

EuroBSDcon 2016 NetBSD Summit

What will we see in this presentation?

ELF, DWARF and `MKDEBUG{,LIB}`

Splitting debug symbols in pkgsrc

Preliminary `SUBPACKAGES` (AKA multi-packages) support

Why. . . . . . not?

> *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."*
>
> – Brian W. Kernighan, Unix for Beginners (1979)

Why?

`gdb -p ‘pgrep tetris‘`

Why?

- Actually in pkgsrc the only way to build packages with debugging symbols is to add appropriate `CFLAGS` and set `INSTALL_UNSTRIPPED` to "yes"

Why?

- Actually in pkgsrc the only way to build packages with debugging symbols is to add appropriate `CFLAGS` and set `INSTALL_UNSTRIPPED` to "yes"
- Debugging symbols can take several disk space, e.g. on NetBSD/amd64 7.99.36:

Why?

- Actually in pkgsrc the only way to build packages with debugging symbols is to add appropriate `CFLAGS` and set `INSTALL_UNSTRIPPED` to "yes"
- Debugging symbols can take several disk space, e.g. on NetBSD/amd64 7.99.36:
  - `{,x}debug.tgz` are 561.672MB (about 1.5GB when extracted)

Why?

- Actually in pkgsrc the only way to build packages with debugging symbols is to add appropriate `CFLAGS` and set `INSTALL_UNSTRIPPED` to "yes"
- Debugging symbols can take several disk space, e.g. on NetBSD/amd64 7.99.36:
  - `{,x}debug.tgz` are 561.672MB (about 1.5GB when extracted)
  - `*.tgz` are 1028.59MB

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)

[1] . . . and a lot of other Unix-like operating systems
[2] Executable and Linkable Format
[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:

[1] . . . and a lot of other Unix-like operating systems
[2] Executable and Linkable Format
[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:
  - ELF file header

[1] . . . and a lot of other Unix-like operating systems
[2] Executable and Linkable Format
[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:
  - ELF file header
  - segments (system loader POV)

[1] . . . and a lot of other Unix-like operating systems
[2] Executable and Linkable Format
[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:
  - ELF file header
  - segments (system loader POV)
  - sections (toolchain POV, also the interesting perspective to handle debug information)

[1] . . . and a lot of other Unix-like operating systems

[2] Executable and Linkable Format

[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:
  - ELF file header
  - segments (system loader POV)
  - sections (toolchain POV, also the interesting perspective to handle debug information)
- Debug information are stored in `.debug_*` sections (in the DWARF [3] format)

[1] . . . and a lot of other Unix-like operating systems

[2] Executable and Linkable Format

[3] Debugging With Attributed Record Formats

How debug information are stored? (ELF, DWARF)

- NetBSD [1] uses the ELF [2] format (executable, relocatable, shared and core are all ELF object files)
- ELF files are basically made up of:
  - ELF file header
  - segments (system loader POV)
  - sections (toolchain POV, also the interesting perspective to handle debug information)
- Debug information are stored in `.debug_*` sections (in the DWARF [3] format)
- `readelf(1)` and `objdump(1)` can be used to display information about ELF and other object format files

[1] . . . and a lot of other Unix-like operating systems
[2] Executable and Linkable Format
[3] Debugging With Attributed Record Formats

A quick look at them via `readelf(1)`: ELF file header

```
$ readelf -h /sbin/init
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1dd0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          34352 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         33
  Section header string table index: 30
```

A quick look at them via `readelf(1)`: segments

```
$ readelf -lW /sbin/init

Elf file type is DYN (Shared object file)
Entry point 0x1dd0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0001c0 0x0001c0 R E 0x8
  INTERP         0x000200 0x0000000000000200 0x0000000000000200 0x000013 0x000013 R   0x1
      [Requesting program interpreter: /libexec/ld.elf_so]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x0056b0 0x0056b0 R E 0x200000
  LOAD           0x005e00 0x0000000000205e00 0x0000000000205e00 0x0004e0 0x000968 RW  0x200000
  DYNAMIC        0x005e28 0x0000000000205e28 0x0000000000205e28 0x0001a0 0x0001a0 RW  0x8
  NOTE           0x000214 0x0000000000000214 0x0000000000000214 0x00002c 0x00002c R   0x4
  GNU_EH_FRAME   0x004e00 0x0000000000004e00 0x0000000000004e00 0x00013c 0x00013c R   0x4
  GNU_RELRO      0x005e00 0x0000000000205e00 0x0000000000205e00 0x000200 0x000200 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.netbsd.ident .note.netbsd.pax .hash .dynsym .dynstr .rela.dyn .rela.plt .init
          .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.netbsd.ident .note.netbsd.pax
   06     .eh_frame_hdr
   07     .ctors .dtors .jcr .dynamic .got
```

A quick look at them via `readelf(1)`: sections

```
$ readelf -SW /sbin/init
There are 33 section headers, starting at offset 0x8630:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000200 000200 000013 00   A  0   0  1
  [ 2] .note.netbsd.ident NOTE            0000000000000214 000214 000018 00   A  0   0  4
  [ 3] .note.netbsd.pax  NOTE            000000000000022c 00022c 000014 00   A  0   0  4
  [ 4] .hash             HASH            0000000000000240 000240 000284 04   A  5   0  8
  [ 5] .dynsym           DYNSYM          00000000000004c8 0004c8 0008a0 18   A  6   2  8
  [ 6] .dynstr           STRTAB          0000000000000d68 000d68 000338 00   A  0   0  1
  [ 7] .rela.dyn         RELA            00000000000010a0 0010a0 000108 18   A  5   0  8
  [ 8] .rela.plt         RELA            00000000000011a8 0011a8 000720 18  AI  5  22  8
  [ 9] .init             PROGBITS        00000000000018d0 0018d0 00000e 00  AX  0   0 16
  [10] .plt              PROGBITS        00000000000018e0 0018e0 0004d0 10  AX  0   0 16
  [11] .plt.got          PROGBITS        0000000000001db0 001db0 000020 00  AX  0   0  8
  [12] .text             PROGBITS        0000000000001dd0 001dd0 002733 00  AX  0   0 16
[...]
  [26] .ident            PROGBITS        0000000000000000 006301 00018d 00      0   0  1
  [27] .copyright        PROGBITS        0000000000000000 00648e 000061 00      0   0  1
  [28] .SUNW_ctf         PROGBITS        0000000000000000 0064f0 0009a9 00      0   0  4
  [29] .gnu_debuglink    PROGBITS        0000000000000000 006e99 000010 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 008523 000109 00      0   0  1
  [31] .symtab           SYMTAB          0000000000000000 006eb0 0010b0 18     32  82  8
  [32] .strtab           STRTAB          0000000000000000 007f60 0005c3 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

MKDEBUG{,LIB}

- NetBSD provides `MKDEBUG` and `MKDEBUGLIB` system variables to split debugging symbols and generate - respectively - `*.debug` and `lib*_g.a` files

MKDEBUG{,LIB}

- NetBSD provides `MKDEBUG` and `MKDEBUGLIB` system variables to split debugging symbols and generate - respectively - `*.debug` and `lib*_g.a` files
- When they are set `debug.tgz` and `xdebug.tgz` installation sets are generated, containing all the split debug symbols

MKDEBUG{,LIB}

- NetBSD provides `MKDEBUG` and `MKDEBUGLIB` system variables to split debugging symbols and generate - respectively - `*.debug` and `lib*_g.a` files
- When they are set `debug.tgz` and `xdebug.tgz` installation sets are generated, containing all the split debug symbols
- `*.debug` files are installed in `/usr/libdata/debug/` directory (`MKDEBUG`)

MKDEBUG{,LIB}

- NetBSD provides `MKDEBUG` and `MKDEBUGLIB` system variables to split debugging symbols and generate - respectively - `*.debug` and `lib*_g.a` files
- When they are set `debug.tgz` and `xdebug.tgz` installation sets are generated, containing all the split debug symbols
- `*.debug` files are installed in `/usr/libdata/debug/` directory (`MKDEBUG`)
- `lib*_g.a` files are installed in the appropriate `lib/` directories (`MKDEBUGLIB`)

`MKDEBUG{,LIB}` under the hood

- `-g` flag is added to the `CFLAGS`

MKDEBUG{,LIB} under the hood

- `-g` flag is added to the `CFLAGS`
- `objcopy --only-keep-debug <file> <file>.debug` is invoked to split the debug symbols from `<file>` to `<file>.debug`

MKDEBUG{,LIB} under the hood

- -g flag is added to the CFLAGS
- objcopy --only-keep-debug <file> <file>.debug is invoked to split the debug symbols from <file> to <file>.debug
- objcopy --strip-debug -p -R .gnu_debuglink
  --add-gnu-debuglink=<file>.debug <file>
  is invoked to:

MKDEBUG{,LIB} under the hood

- -g flag is added to the CFLAGS
- objcopy --only-keep-debug <file> <file>.debug is invoked to split the debug symbols from <file> to <file>.debug
- objcopy --strip-debug -p -R .gnu_debuglink
  --add-gnu-debuglink=<file>.debug <file>
  is invoked to:
  - -p is used preserve the dates (access and modification dates will be the same for <file> and <file>.debug)

MKDEBUG{,LIB} under the hood

- -g flag is added to the CFLAGS
- objcopy --only-keep-debug <file> <file>.debug is invoked to split the debug symbols from <file> to <file>.debug
- objcopy --strip-debug -p -R .gnu_debuglink
  --add-gnu-debuglink=<file>.debug <file>
  is invoked to:
  - -p is used preserve the dates (access and modification dates will be the same for <file> and <file>.debug)
  - -R .gnu_debuglink is used to remove any already existing .gnu_debuglink ELF section

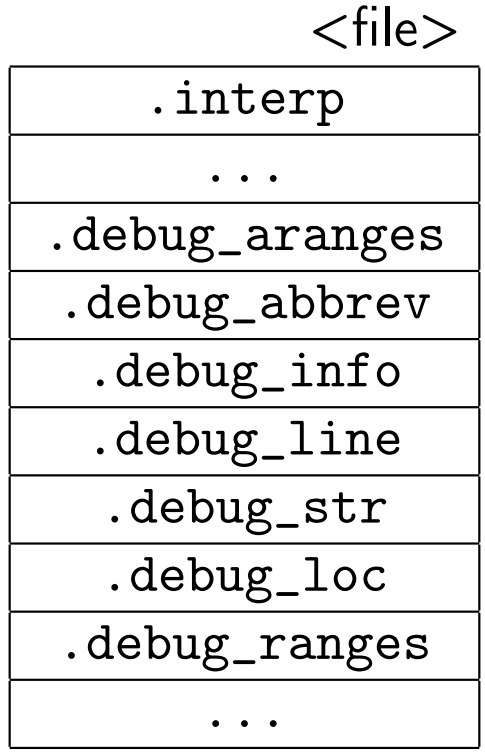
MKDEBUG{,LIB} under the hood

- `-g` flag is added to the `CFLAGS`
- `objcopy --only-keep-debug <file> <file>.debug` is invoked to split the debug symbols from `<file>` to `<file>.debug`
- `objcopy --strip-debug -p -R .gnu_debuglink --add-gnu-debuglink=<file>.debug <file>` is invoked to:
  - `-p` is used preserve the dates (access and modification dates will be the same for `<file>` and `<file>.debug`)
  - `-R .gnu_debuglink` is used to remove any already existing `.gnu_debuglink` ELF section
  - `--add-gnu-debuglink=<file>.debug` is used to create a reference to the corresponding `*.debug` file (only the `basename(1)` is honored)

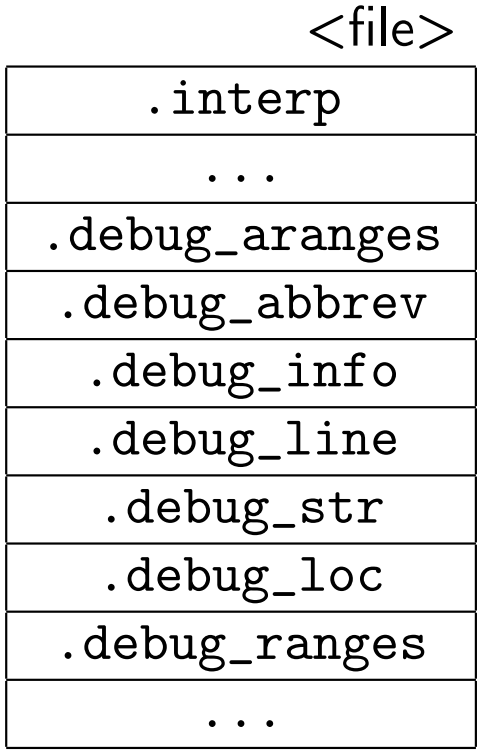
MKDEBUG{,LIB} under the hood

- -g flag is added to the CFLAGS
- objcopy --only-keep-debug <file> <file>.debug is invoked to split the debug symbols from <file> to <file>.debug
- objcopy --strip-debug -p -R .gnu_debuglink
  --add-gnu-debuglink=<file>.debug <file>
  is invoked to:
  - -p is used preserve the dates (access and modification dates will be the same for <file> and <file>.debug)
  - -R .gnu_debuglink is used to remove any already existing .gnu_debuglink ELF section
  - --add-gnu-debuglink=<file>.debug is used to create a reference to the corresponding *.debug file (only the basename(1) is honored)
  - --strip-debug strip all the debug sections in <file>

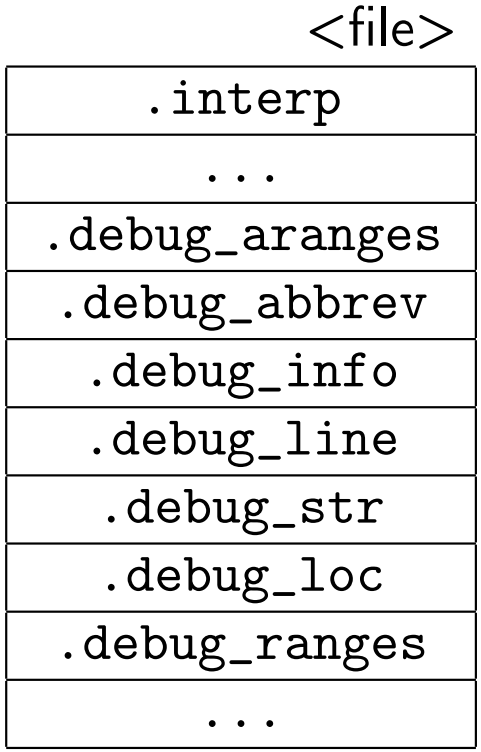
MKDEBUG{,LIB} under the hood (illustrated): <file> compiled with debugging flags



MKDEBUG{,LIB} under the hood (illustrated): generation of <file>.debug

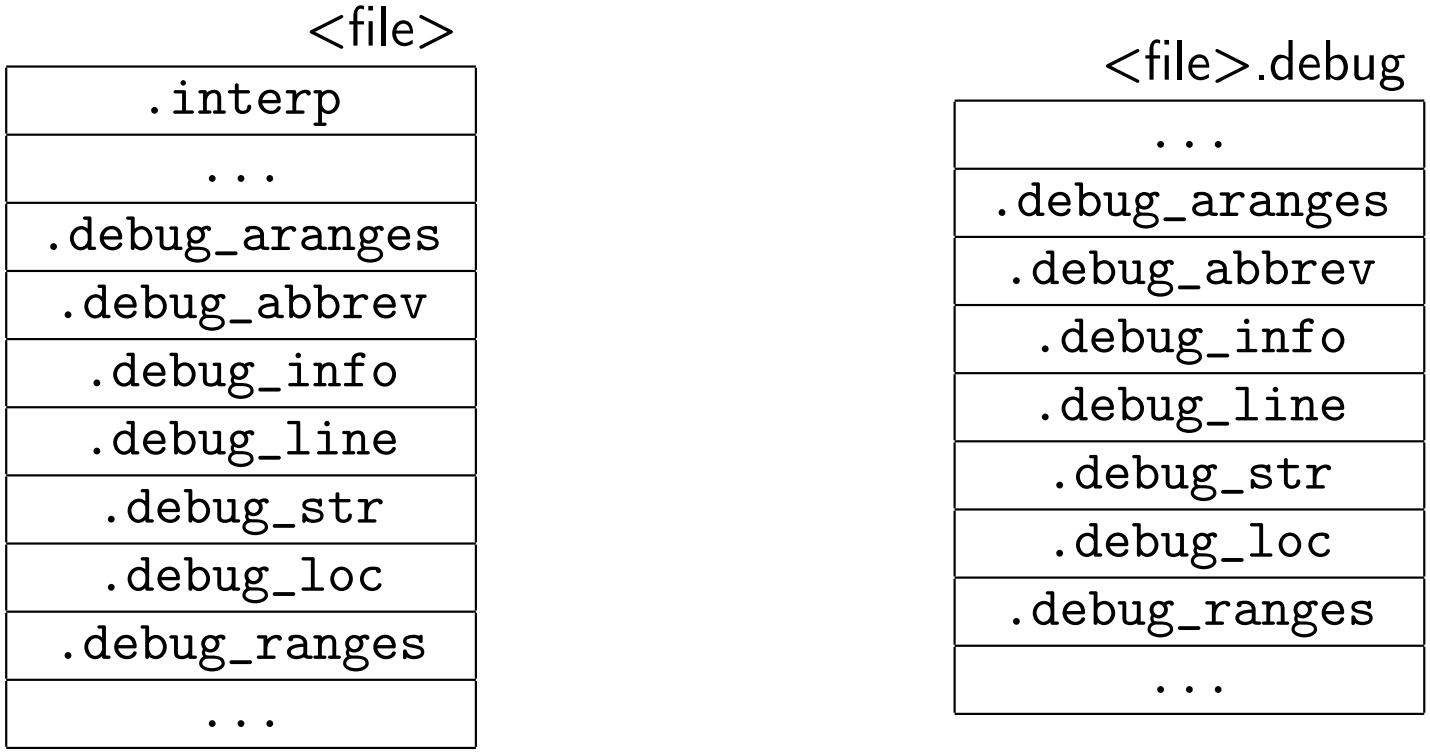


MKDEBUG{,LIB} under the hood (illustrated): generation of <file>.debug



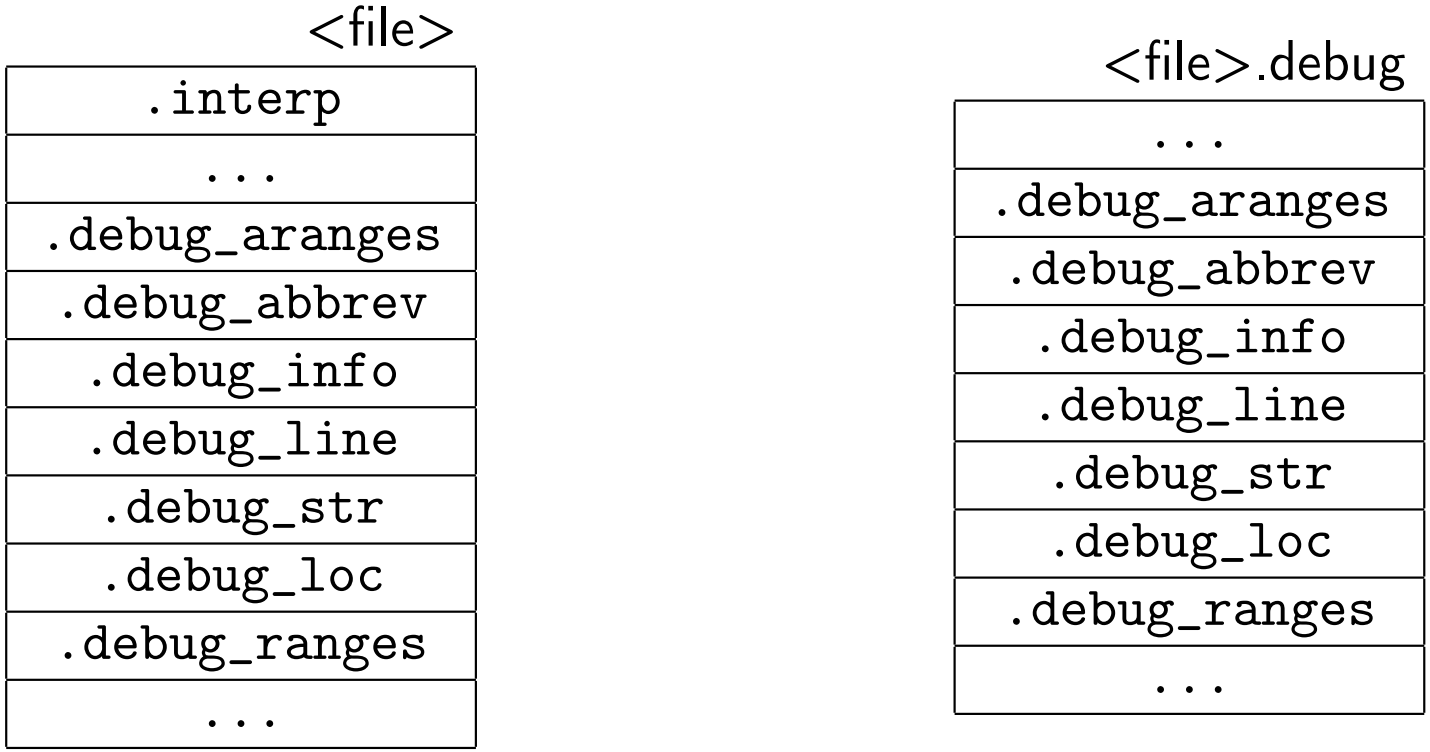

```
objcopy --only-keep-debug <file> \
    <file>.debug
```

MKDEBUG{,LIB} under the hood (illustrated): generation of <file>.debug

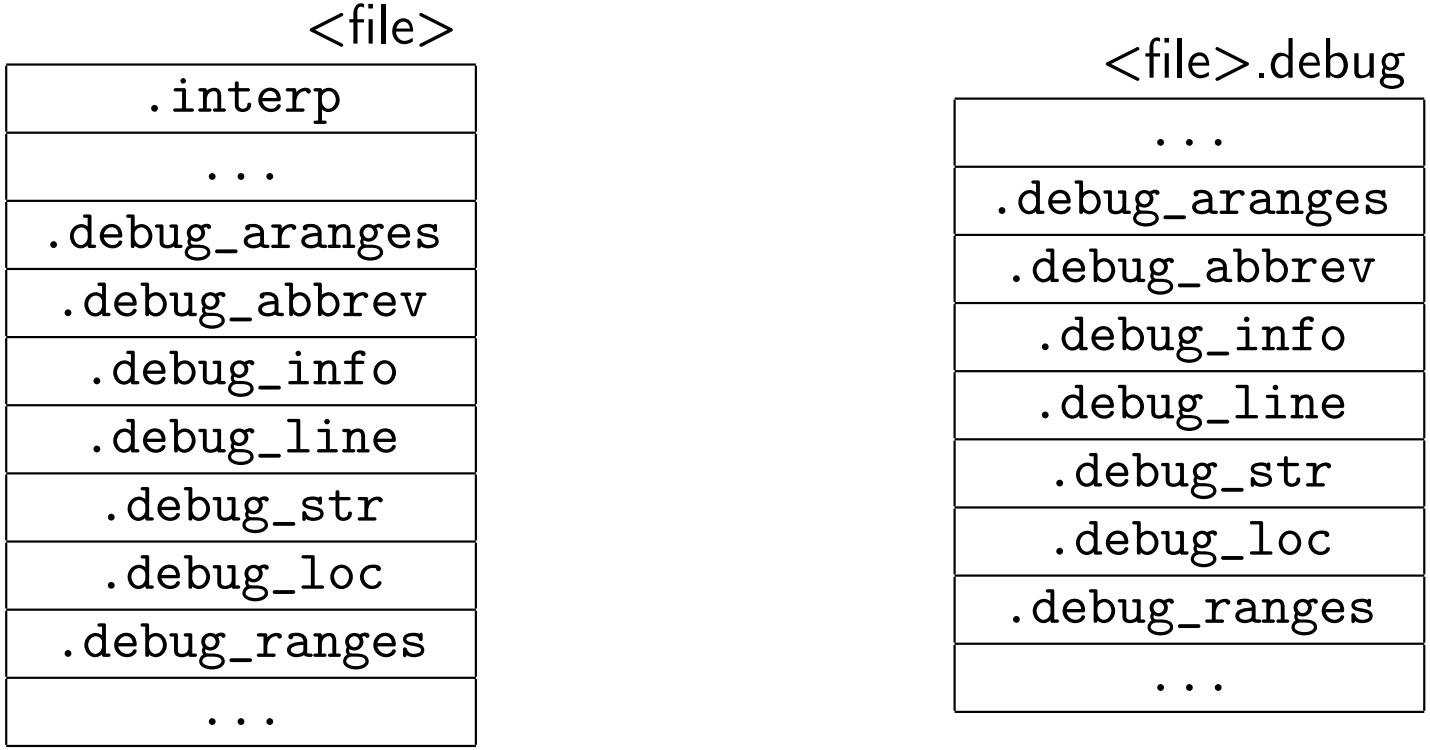



```
objcopy --only-keep-debug <file> \
    <file>.debug
```

MKDEBUG{,LIB} under the hood (illustrated): stripping of <file>

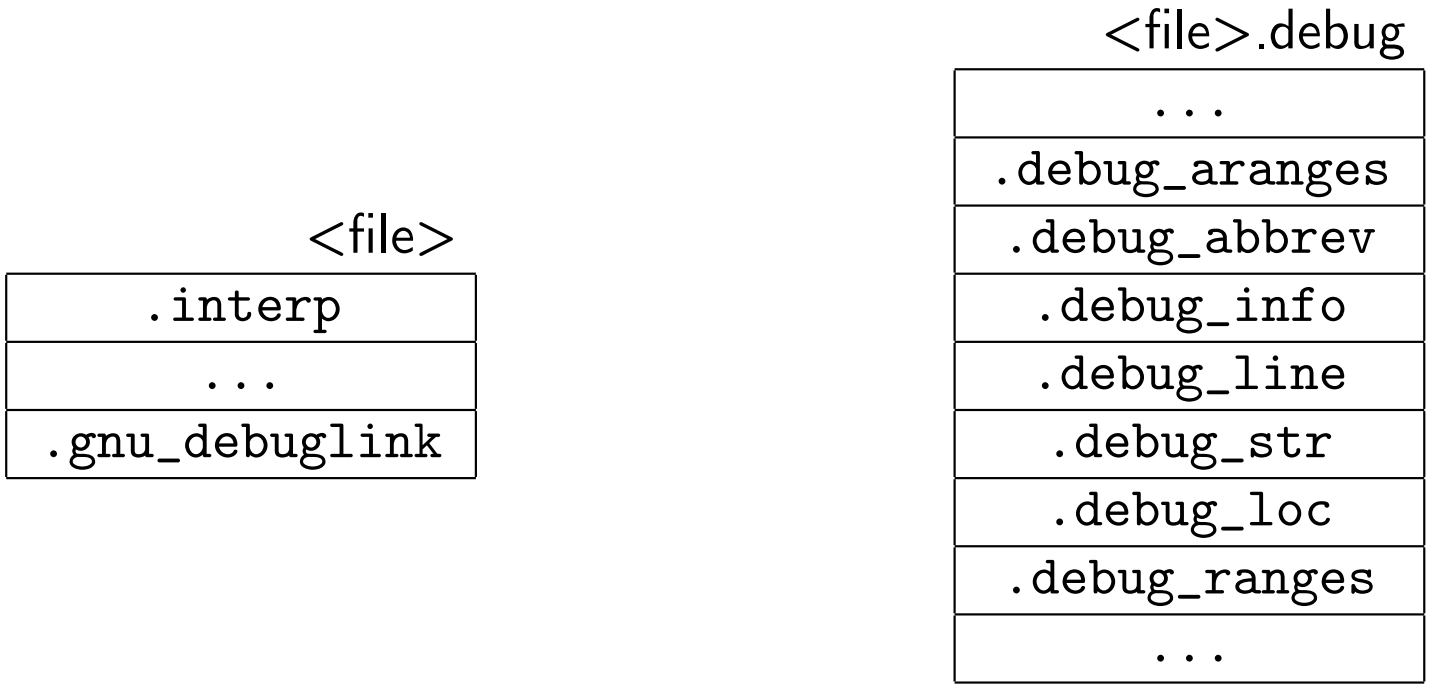


MKDEBUG{,LIB} under the hood (illustrated): stripping of <file>




```
objcopy --strip-debug -p -R .gnu_debuglink \
    --add-gnu-debuglink=<file>.debug <file>
```

MKDEBUG{,LIB} under the hood (illustrated): stripping of <file>




```
objcopy --strip-debug -p -R .gnu_debuglink \
    --add-gnu-debuglink=<file>.debug <file>
```

Splitting debug symbols in pkgsrc: `bsd.debugdata.mk`

- `bsd.debugdata.mk` implements stripping of the debug data from package's programs/libraries

Splitting debug symbols in pkgsrc: `bsd.debugdata.mk`

- `bsd.debugdata.mk` implements stripping of the debug data from package's programs/libraries
- Works similarly to `MKDEBUG{,LIB}` after the `post-install` phase

Splitting debug symbols in pkgsrc: `bsd.debugdata.mk`

- `bsd.debugdata.mk` implements stripping of the debug data from package's programs/libraries
- Works similarly to `MKDEBUG{,LIB}` after the `post-install` phase
- `*.debug` files are dynamically appended to the package's `PLIST`

Splitting debug symbols in pkgsrc: `bsd.debugdata.mk`

- `bsd.debugdata.mk` implements stripping of the debug data from package's programs/libraries
- Works similarly to `MKDEBUG{,LIB}` after the `post-install` phase
- `*.debug` files are dynamically appended to the package's `PLIST`
- Turned on if `PKG_DEBUGDATA` is "yes"

Splitting debug symbols in pkgsrc: `bsd.debugdata.mk`

- `bsd.debugdata.mk` implements stripping of the debug data from package's programs/libraries
- Works similarly to `MKDEBUG{,LIB}` after the `post-install` phase
- `*.debug` files are dynamically appended to the package's `PLIST`
- Turned on if `PKG_DEBUGDATA` is "yes"
- Granularity of debug information can be adjusted via `PKG_DEBUGLEVEL` ("small", "default" or "detailed")

Splitting debug symbols in pkgsrc: `check/check-debugdata.mk`

- Performs various sanity checks about debugdata:

Splitting debug symbols in pkgsrc: `check/check-debugdata.mk`

- Performs various sanity checks about debugdata:
  - Check that every program/library has a corresponding `.debug` file

Splitting debug symbols in pkgsrc: `check/check-debugdata.mk`

- Performs various sanity checks about debugdata:
  - Check that every program/library has a corresponding `.debug` file
  - Check for `.gnu_debuglink` ELF section in every program/library

Splitting debug symbols in pkgsrc: `check/check-debugdata.mk`

- Performs various sanity checks about debugdata:
  - Check that every program/library has a corresponding `.debug` file
  - Check for `.gnu_debuglink` ELF section in every program/library
  - Warn if `.debug` file does not contain a `.debug_info` ELF section

Preliminary SUBPACKAGES (AKA multi-packages) support

- `*.debug` files should be installed in a separate package (e.g. for `<package>`, `<package>-debug`)

Preliminary SUBPACKAGES (AKA multi-packages) support

- `*.debug` files should be installed in a separate package (e.g. for `<package>`, `<package>-debug`)
- Multi-package support is needed to do that

Preliminary SUBPACKAGES (AKA multi-packages) support

- *.debug files should be installed in a separate package (e.g. for <package>, <package>-debug)
- Multi-package support is needed to do that
- From MAINTAINER POV some variables and files will become per-SUBPACKAGES (e.g.: COMMENT.<spkg>, PLIST.<spkg>, etc.)

Preliminary SUBPACKAGES (AKA multi-packages) support

- *.debug files should be installed in a separate package (e.g. for <package>, <package>-debug)
- Multi-package support is needed to do that
- From MAINTAINER POV some variables and files will become per-SUBPACKAGES (e.g.: COMMENT.<spkg>, PLIST.<spkg>, etc.)
- At the moment that is mostly implemented duplicating existing logic, i.e.:

```
.if !empty(SUBPACKAGES)
  .for _spkg_ in ${SUBPACKAGES}
    <subpackages logic>
  .endfor
.else # !SUBPACKAGES
  <non-subpackages (i.e. already existent) logic>
.endif # SUBPACKAGES
```

Preliminary SUBPACKAGES (AKA multi-packages) support

- . . . but that's still far from complete! (preliminary support in `mk/plist/*`, `mk/pkgformat/*/*` and `mk/check/*`. . . `mk/pkginstall/*` and other parts of `mk/*` still completely unaware of SUBPACKAGES existence!)

Conclusion/TODOs

- Complete `SUBPACKAGES` support (via code duplicated logic)

Conclusion/TODOs

- Complete `SUBPACKAGES` support (via code duplicated logic)
- Add implicit (and hidden) subpackage, in other words: every package will always have at least one subpackage (this will permit to get rid of code duplication and have a single control flow)

Conclusion/TODOs

- Complete `SUBPACKAGES` support (via code duplicated logic)
- Add implicit (and hidden) subpackage, in other words: every package will always have at least one subpackage (this will permit to get rid of code duplication and have a single control flow)
- Adapt `mk/bsd.debugdata.mk` to `SUBPACKAGES`

Thanks

- Google for organizing Google Summer of Code

Thanks

- Google for organizing Google Summer of Code
- The NetBSD Foundation

Thanks

- Google for organizing Google Summer of Code
- The NetBSD Foundation
- My mentors: David Maxwell (<david>), Jörg Sonnenberger (<joerg>), Taylor R. Campbell (<riastradh>), Thomas Klausner (<wiz>), William J. Coldwell (<billc>)

Thanks

- Google for organizing Google Summer of Code
- The NetBSD Foundation
- My mentors: David Maxwell (<david>), Jörg Sonnenberger (<joerg>), Taylor R. Campbell (<riastradh>), Thomas Klausner (<wiz>), William J. Coldwell (<billc>)
- . . . and everyone that helped via #netbsd-code, MLs or private emails, in particular Kamil Rytarowski (<kamil>) and Christos Zoulas (<christos>)

References I

Eric Youngdale.
The ELF Object File Format: Introduction.
https://www.linuxjournal.com/article/1059, a.

Eric Youngdale.
The ELF Object File Format by Dissection.
https://www.linuxjournal.com/article/1060, b.

Michael Eager.
Introduction to the DWARF Debugging Format.
http://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf.

TIS Committee.
Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2.
http://refspecs.linuxbase.org/elf/elf.pdf.

References II

DWARF Debugging Information Format Committee.
DWARF Debugging Information Format Version 4.
http://dwarfstd.org/doc/DWARF4.pdf.

Samy Al Bahra.
!!Con 2016 - Debugging debuggers!!!
https://www.youtube.com/watch?v=OEaOEfJja_Y.

Peter Jay Salzman.
Using GNU's GDB Debugger.
http://dirac.org/linux/gdb/.

Free Software Foundation, Inc.
Debugging with GDB Tenth Edition for GDB Version 7.10.1.
http://sourceware.org/gdb/download/onlinedocs/gdb/index.html.

References III

Mark J. Wielaard.
Where are your Symbols, Debuginfo and Sources?
https://gnu.wildebeest.org/blog/mjw/2016/02/02/where-are-your-symbols-debuginfo-and-sources/.

The pkgsrc Developers Alistair Crooks, Hubert Feyrer.
The pkgsrc guide.
https://www.netbsd.org/docs/pkgsrc/.