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
“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”

– Brian W. Kernighan, Unix for Beginners (1979)
Why?

Score: 102

paused - press RETURN to continue

Reading symbols from /usr/libexec/ld.elf.so...Reading symbols from /usr/libexec/ld.elf_so.debug...done.
done.
0x000079c3f8e3dc1a in poll () from /usr/lib/libc.so.12
(gdb) set score = 12345678
(gdb) cont

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 uses the ELF 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 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
2Executable and Linkable Format
3Debugging 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileSiz</th>
<th>MemSiz</th>
<th>Flg</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHDR</td>
<td>0x000040</td>
<td>0x0000000000000040</td>
<td>0x0000000000000040</td>
<td>0x0001c0</td>
<td>0x0001c0</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td>INTERP</td>
<td>0x000200</td>
<td>0x0000000000000200</td>
<td>0x0000000000000200</td>
<td>0x000013</td>
<td>0x000013</td>
<td>R</td>
<td>0x1</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0056b0</td>
<td>0x0056b0</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x005e00</td>
<td>0x00000000000205e00</td>
<td>0x00000000000205e00</td>
<td>0x0004e0</td>
<td>0x000968</td>
<td>RW</td>
<td>0x200000</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>0x005e28</td>
<td>0x00000000000205e28</td>
<td>0x00000000000205e28</td>
<td>0x0001a0</td>
<td>0x0001a0</td>
<td>RW</td>
<td>0x8</td>
</tr>
<tr>
<td>NOTE</td>
<td>0x000214</td>
<td>0x0000000000000214</td>
<td>0x0000000000000214</td>
<td>0x00002c</td>
<td>0x00002c</td>
<td>R</td>
<td>0x4</td>
</tr>
<tr>
<td>GNU_EH_FRAME</td>
<td>0x004e00</td>
<td>0x000000000004e00</td>
<td>0x000000000004e00</td>
<td>0x00013c</td>
<td>0x00013c</td>
<td>R</td>
<td>0x4</td>
</tr>
<tr>
<td>GNU_RELRO</td>
<td>0x005e00</td>
<td>0x00000000000205e00</td>
<td>0x00000000000205e00</td>
<td>0x000200</td>
<td>0x000200</td>
<td>R</td>
<td>0x1</td>
</tr>
</tbody>
</table>

Section to Segment mapping:

Segment Sections...

00 .interp
01 .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`

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

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>0</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.interp</td>
<td>PROGBITS</td>
<td>0000000000</td>
<td>200</td>
<td>200</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>.note.netbsd.ident</td>
<td>NOTE</td>
<td>0000000000000214</td>
<td>000214</td>
<td>000018</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>.note.netbsd.pax</td>
<td>NOTE</td>
<td>000000000000022c</td>
<td>00022c</td>
<td>000014</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>.hash</td>
<td>HASH</td>
<td>0000000000000240</td>
<td>000240</td>
<td>000284</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>.dynsym</td>
<td>DYNSYM</td>
<td>00000000000004c8</td>
<td>0004c8</td>
<td>0008a0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>.dynstr</td>
<td>STRTAB</td>
<td>000000000000d68</td>
<td>00d68</td>
<td>00338</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>.rela.dyn</td>
<td>RELA</td>
<td>00000000000010a0</td>
<td>0010a0</td>
<td>000108</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>.rela.plt</td>
<td>RELA</td>
<td>00000000000011a8</td>
<td>0011a8</td>
<td>00720</td>
<td>18</td>
<td>A</td>
<td>5</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>.init</td>
<td>PROGBITS</td>
<td>00000000000018d0</td>
<td>0018d0</td>
<td>0000e</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>.plt</td>
<td>PROGBITS</td>
<td>00000000000018e0</td>
<td>0018e0</td>
<td>0040d0</td>
<td>10</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>.plt.got</td>
<td>PROGBITS</td>
<td>000000000001db0</td>
<td>001db0</td>
<td>00020</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>.text</td>
<td>PROGBITS</td>
<td>000000000000dd0</td>
<td>001dd0</td>
<td>002733</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>26</td>
<td>.ident</td>
<td>PROGBITS</td>
<td>0000000000000000</td>
<td>006301</td>
<td>0018d0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>.copyright</td>
<td>PROGBITS</td>
<td>0000000000000000</td>
<td>00648e</td>
<td>00061</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>.SUNW_ctf</td>
<td>PROGBITS</td>
<td>0000000000000000</td>
<td>0064f0</td>
<td>009a9</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>.gnu_debuglink</td>
<td>PROGBITS</td>
<td>0000000000000000</td>
<td>006e99</td>
<td>00010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>0000000000000000</td>
<td>08523</td>
<td>00109</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>0000000000000000</td>
<td>00eb0</td>
<td>010b0</td>
<td>18</td>
<td>32</td>
<td>82</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>0000000000000000</td>
<td>07f60</td>
<td>005c3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)
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`).
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`
  - `-p` is used to 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 is honored)
- `--strip-debug` strips all the debug sections in `<file>`
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 to 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

- -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 to 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 to 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 to 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

```
<file>

.interp
...
.debug_aranges
.debug_abbrev
.debug_info
.debug_line
.debug_str
.debug_loc
.debug_ranges
...
```
MKDEBUG{,LIB} under the hood (illustrated): generation of `<file>.debug`

```
<file>
  .interp
  ...
  .debug_aranges
  .debug_abbrev
  .debug_info
  .debug_line
  .debug_str
  .debug_loc
  .debug_ranges
  ...
```

```bash
objcopy --only-keep-debug <file> <file>.debug
```
MKDEBUG{,LIB} under the hood (illustrated): generation of `<file>.debug`

```
<file>

.interp
...
.debug_aranges
.debug_abbrev
.debug_info
.debug_line
.debug_str
.debug_loc
.debug_ranges
...
```

```
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>`

<table>
<thead>
<tr>
<th><code>&lt;file&gt;</code></th>
<th><code>&lt;file&gt;.debug</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.interp</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>.debug_aranges</code></td>
<td><code>.debug_aranges</code></td>
</tr>
<tr>
<td><code>.debug_abbrev</code></td>
<td><code>.debug_abbrev</code></td>
</tr>
<tr>
<td><code>.debug_info</code></td>
<td><code>.debug_info</code></td>
</tr>
<tr>
<td><code>.debug_line</code></td>
<td><code>.debug_line</code></td>
</tr>
<tr>
<td><code>.debug_str</code></td>
<td><code>.debug_str</code></td>
</tr>
<tr>
<td><code>.debug_loc</code></td>
<td><code>.debug_loc</code></td>
</tr>
<tr>
<td><code>.debug_ranges</code></td>
<td><code>.debug_ranges</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>
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 \texttt{<file>}

\begin{itemize}
  \item \texttt{.interp}
  \item \ldots
  \item \texttt{.gnu_debuglink}
\end{itemize}

\begin{itemize}
  \item \texttt{.debug}
  \item \ldots
  \item \texttt{.debug_aranges}
  \item \texttt{.debug_abbrev}
  \item \texttt{.debug_info}
  \item \texttt{.debug_line}
  \item \texttt{.debug_str}
  \item \texttt{.debug_loc}
  \item \texttt{.debug_ranges}
  \item \ldots
\end{itemize}

\texttt{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

*Note:*.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: 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>)

...and everyone that helped via #netbsd-code, MLs or private emails, in particular Kamil Rytarowski (<kamil>) and Christos Zoulas (<christos>)
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

Eric Youngdale.
The ELF Object File Format: Introduction.

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

Michael Eager.
Introduction to the DWARF Debugging Format.

TIS Committee.
References II

DWARF Debugging Information Format Committee.

Samy Al Bahra.
!!Con 2016 - Debugging debuggers!!!
https://www.youtube.com/watch?v=OEp0EfJja_Y.

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

Free Software Foundation, Inc.
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/.