기반 작업
(4) struct gcc_debug_hooks 이란?

정원교

2004년 2월 23일

목 차

제 1 절 10 주째 강의를 시작하며

벌써 GCC 강의를 시작한지 10 주째를 맞이하고 있습니다. 그 동안 나름대로 열심히 작성할려고 했는데, 저의 지식 부족도 있고 다른 회사일 때문인 것도 있고 해서 "장인 정신"으로 시작했던 이 일에 많은 지식과 노력을 못 담은 듯해서 안타깝습니다. 다시 초심으로 돌아가 더욱 더 알찬 강의를 작성하도록 하겠습니다. 10 주째 강의를 시작하도록 하겠습니다.

제 2 절 GCC 속의 debug hook 들과 debug format

제가 지금까지 작성한 문서들과 마찬가지로 먼저 역활을 언급하고, 원형을 본 후 각 구성요소의 쓰임새에 대해서 말하는 방식으로 진행하겠습니다.

이 구조체는 command line option 들에 따라 $prefix/gcc/toplev.c 에서 설정되는 global instance debug_hooks 에 따라 달리 접근되어지는 debug information output 을 위한 hook 들을 포함하고 있습니다.

우선 GCC 속에 존재하는 debug format 을 위한 hook 들이 우선 어떤게 있는 부터 살펴보도록 하겠습니다. 이에 대한 정보는 $prefix/gcc/debug.h 파일의 하단 부분에 보시면 알 수 있습니다.

```
extern struct gcc_debug_hooks do_nothing_debug_hooks;
extern struct gcc_debug_hooks dbx_debug_hooks;
extern struct gcc_debug_hooks sdb_debug_hooks;
extern struct gcc_debug_hooks xcoff_debug_hooks;
extern struct gcc_debug_hooks dwarf_debug_hooks;
extern struct gcc_debug_hooks dwarf2_debug_hooks;
extern struct gcc_debug_hooks vmsdbg_debug_hooks;
```

이름이 알려진 debugger 로는 GNU 의 gdb, AIX 의 adb, JAVA 의 JDB, DBX, XDB, WDB, Ladebug, 등등 많이 있습니다. 하지만 각각의 debugger 들은 인식할 수 있는 debug format 이 정해져 있는 경우도 있고 아니면 여러가지의 debug foramt 을 인식하는 것도 있지만, 어쨌든 debug format 이란 것이 있습니다. 그리고 그의 종류는 여러가지가 있습니다.

GNU C compiler 는 '.c' 형태의 C source 를 '.s' 형태의 어셈블리 언어로 바꾸게 되고 어셈블러에 의해서 '.o' 형식으로 바뀌게 됩니다. 그리고 마지막으로 linker 에 의해서 실행 파일로 묶이게 되는데, debug format 이 영향을 미치는 부분은 '.c' 형태가 '.s' 로 바뀔 때 영향을 미치게 됩니다.

GCC 를 컴파일할 때 '-g' 옵션을 사용해서 컴파일하게 되면 '.s' 파일에는 추가적인 debug 정보들이 들어가게 됩니다. 이러한 debug 정보들에는 source code 의 줄 번호, type, 변수의 영향 범위, 함수 이름들, parameter 들 등등의 여러 정보가 들어가게 됩니다.

debug format 을 어떻게 설정하느냐에 따라서 어떤 형식의 문법을 사용할 지가 결정되게 됩니다. GCC 옵션 중 debug 에 해당하는 부분에 대한 세부 설명은 gcc man page 를 참고하시기 바랍니다.

위에서 나열한 hook 들에 대해서 좀 설명하겠습니다.

- `do_nothing_debug_hooks`
  debug 정보를 생성하지 않을 경우나 struct gcc_debug_hooks 의 기본값으로 설정되는 것입니다.
- `dbx_debug_hooks`
  이름에서도 알수 있듯이 DBX debug format 을 생성합니다. stabs format 을 생성할 경우 이 hook 을 사용하게 됩니다.
- `sdb_debug_hooks`
  COFF format 을 위한 debug 정보를 생성합니다.
- `xcoff_debug_hooks`
  XCOFF format 을 위한 debug 정보를 생성합니다.
- `dwarf_debug_hooks`
  DWARF version 1 을 위한 debug 정보를 생성합니다.
- `dwarf2_debug_hooks`
  DWARF version 2 르 위한 debug 정보를 생성합니다.
- `vmsdbg_debug_hooks`
  VMS debug format 형태로 debug 정보를 생성합니다.

각각의 hook 에 대한 설명을 하였으니 실제로 어디서 debug hooks 에 대한 정보를 가지고 있고 각 DEBUG format 정보는 또한 어디에 있으며, 어떻게 구성되는지 그리고 각 hook 들의 역활에 대해 차례차례 알아보도록 하겠습니다.

제 3 절 struct gcc_debug_hooks 의 구조

GCC 에서 결과적으로 debug hook 를 포함하는 것은 $prefix/gcc/toplev.c 파일에 선언되어 있는 아래 변수입니다.

```
struct gcc_debug_hooks *debug_hooks = &do_nothing_debug_hooks;
```

기본값으로 위에서 언급한 do_nothing_debug_hooks 을 가지고 있습니다. 이 hook 에 대해서 알아 보기 전에 struct gcc_debug_hooks 이 어떻게 구조화되어 있는지 알아봅시다.

먼저 이 구조체의 원형을 보도록 하겠습니다. $prefix/gcc/debug.h 파일에 해당 구조체가 선언되어 있습니다. 아래와 같습니다.

```
struct gcc_debug_hooks
{
  void (* init) PARAMS ((const char *main_filename));
  void (* finish) PARAMS ((const char *main_filename));
  void (* define) PARAMS ((unsigned int line, const char *text));
  void (* undef) PARAMS ((unsigned int line, const char *macro));
  void (* start_source_file)
                         PARAMS ((unsigned int line, const char *file));
  void (* end_source_file) PARAMS ((unsigned int line));
  void (* begin_block) PARAMS ((unsigned int line, unsigned int n));
  void (* end_block) PARAMS ((unsigned int line, unsigned int n));
  bool (* ignore_block) PARAMS ((tree));
  void (* source_line) PARAMS ((unsigned int line, const char *file));
  void (* begin_prologue) PARAMS ((unsigned int line, const char *file));
  void (* end_prologue) PARAMS ((unsigned int line));
  void (* end_epilogue) PARAMS ((void));
  void (* begin_function) PARAMS ((tree decl));
  void (* end_function) PARAMS ((unsigned int line));
  void (* function_decl) PARAMS ((tree decl));
  void (* global_decl) PARAMS ((tree decl));
  void (* deferred_inline_function) PARAMS ((tree decl));
  void (* outlining_inline_function) PARAMS ((tree decl));
  void (* label) PARAMS ((rtx));
};
```

그렇게 복잡한 역활을 하는 구조체는 아니지만 모두가 함수 포인터를 가지고 있기 때문에 각 함수 포인터가 무엇과 연결되는지에 대해 관심을 가져야 할 것입니다.

각 구성 요소에 대한 설명을 하도록 하겠습니다.

- `* init`
  debug output 을 초기화합니다. MAIN_FILENAME 은 main 입력 파일을 이름입니다.
- `* finish`
  debug symbol 들을 output 합니다.
- `* define`
  이름과 확장 TEXT 를 가지는 줄 LINE 상에 define 된 매크로.
- `* undef`
  줄 LINE 에서 undefine 된 매크로.
- `* start_source_file`
  이전 것의 LINE 번호로부터 새로운 원시 파일 FILE 의 시작을 기록합니다.
- `* end_source_file`
  원시 파일의 회수 (재개시 : resumption) 를 기록합니다. LINE 은 우리가 반환하고자 하는 원시 파일에서의 줄 번호입니다.
- `* begin_block`
  LINE 에서의 Block N 의 시작을 기록합니다. 이것은 1 부터 셈하고
  Function-scope block 을 포함하지 않습니다.

- `* end_block`
  Block 의 끝을 기록합니다. begin_block 으로써의 인자들.

- `* ignore_block`
  만약 현재 BLOCK 이 어떤한 명령어들을 포함하지 않기에 BLOCK 을 위한 어떠한 debugging 정보를 emit 하는 것이 적당하지 않는다면 0 이 아닌 값을 반환합니다. 이것은 nested 함수들을 포함하는 block 들에는 해당이 안될 수 있습니다. 왜냐하면 비록 BLOCK 정보가 더럽혀졌다 (messed up) 하더라도 그러한 함수가 실제로 호출될 수 있기 때문입니다. 기본값은 true 입니다.

- `* source_line`
  (FILE, LINE) 에서의 원시 파일 위치를 기록합니다.

- `* begin_prologue`
  Prologue code 의 시작 부분에서 호출됩니다. LINE 은 함수내에서의 처음 줄을 나타냅니다. 이것은 source_line 과 같은 prototype 을 가지고 있습니다. 그래서 source_line hook 이 필요할 경우 대체될 수 있습니다.

- `* end_prologue`
  Prologue code 의 끝 부분에서 호출됩니다. LINE 은 함수내에서의 처음 줄을 나타냅니다.

- `* end_epilogue`
  Epilogue code 의 끝을 기록합니다.

- `* begin_function`
  함수 DECL 가 선언되기 전에 함수 DECL 의 시작 부분에 호출됩니다.

- `* end_function`
  함수의 끝을 기록합니다. LINE 은 함수내에서의 가장 큰 줄 번호입니다.

- `* function_decl`
  함수 DECL 을 위한 debug information. 이것은 함수의 이름 (symbol) 과 함수의 parameter, 또 함수의 body 를 구성하는 block, 그리고 함수의 지역 변수들을 포함합니다.

- `* global_decl`
  Gobal DECL 을 위한 debug information. Compilation proper 가 마무리된 후 toplev.c 에서 호출됩니다.

- `* deferred_inline_function`
  DECL 는 body 가 존재하는 inline 함수입니다. 하지만 이 시점에서는 output 되지 않았습니다.

- `* outlining_inline_function`
  DECL 는 줄(line) 밖으로 emit 된 것에 관한 inline 함수입니다. Hook 을 사용하는 것은 상당히 쓸모가 있는데, 예를 들면 inline 이 최적화에 의해 엉망으로 되기 전에 추상적인 debug info 를 emit 할때 유용할 수 있습니다.

- `* label`
  LABEL_NAME 이 null 이 아닌 어떤 CODE_LABEL 명령어를 위해 final_scan_insn 로부터 호출됩니다.

제 4 절 각 debug hook 에 대한 기본값

위에서는 구조체에 대한 모습을 봤으므로 여기에서는 각 hook 들이 어떤 값을 가지고 있는지에 대해 보도록 하겠습니다. 하지만 각각의 함수가 어떠한 기능으로 돌아가는지에 대한 세부 사항은 적지 않습니다.

4.1 do_nothing_debug_hooks

아무 일도 하지 않는 debug hook 들을 가집니다. 이 정의는 $prefix/gcc/debug.c 파일에 선언되어 있습니다. 그리고 여기서 지정된 hook 들은 이 파일 밑에 선언되어 있습니다. 원형은 아래와 같습니다.

```
struct gcc_debug_hooks do_nothing_debug_hooks =
{
  debug_nothing_charstar,
  debug_nothing_charstar,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  debug_nothing_int,
  debug_nothing_int_int,
  debug_nothing_int_int,
  debug_true_tree,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  debug_nothing_int,
  debug_nothing_void,
  debug_nothing_tree,
  debug_nothing_int,
  debug_nothing_tree,
  debug_nothing_tree,
  debug_nothing_tree,
  debug_nothing_tree,
  debug_nothing_rtx
};
```

4.2 dbx_debug_hooks

이 정의는 $prefix/gcc/dbxout.c 파일에 정의되어 있습니다.

```
struct gcc_debug_hooks dbx_debug_hooks =
{
  dbxout_init,
  dbxout_finish,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  dbxout_start_source_file,
  dbxout_end_source_file,
  dbxout_begin_block,
  dbxout_end_block,
  debug_true_tree,
  dbxout_source_line,
  dbxout_source_line,
  debug_nothing_int,
  debug_nothing_void,
#ifdef DBX_FUNCTION_FIRST
  dbxout_begin_function,
#else
  debug_nothing_tree,
#endif
```

```
  debug_nothing_int,
  dbxout_function_decl,
  dbxout_global_decl,
  debug_nothing_tree,
  debug_nothing_tree,
  debug_nothing_rtx
};
```

저의 환경에서는 $prefix/gcc/config/dbxelf.h 파일에 선언되어
있는 DBX_FUNCTION_FIRST 정의에 의해서 dbxout_begin_function 가 선택되어집니다.

- DBX_FUNCTION_FIRST
  이것이 작동하게 만들기 위해서는 함수들은 줄 정보 이전에 반드시 나타나야 합니다.

4.3 sdb_debug_hooks

이 정의는 $prefix/gcc/sdbout.c 파일에 정의되어 있습니다.

```
struct gcc_debug_hooks sdb_debug_hooks =
{
  sdbout_init,
  sdbout_finish,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  sdbout_start_source_file,
  sdbout_end_source_file,
  sdbout_begin_block,
  sdbout_end_block,
  debug_true_tree,
  sdbout_source_line,
#ifdef MIPS_DEBUGGING_INFO
  debug_nothing_int_charstar,
  sdbout_end_prologue,
#else
  sdbout_begin_prologue,
  debug_nothing_int,
#endif
  sdbout_end_epilogue,
  sdbout_begin_function,
  sdbout_end_function,
  debug_nothing_tree,
  sdbout_global_decl,
  debug_nothing_tree,
  debug_nothing_tree,
  sdbout_label
};
```

위에서 볼 수 있는 MIPS_DEBUGGING_INFO 매크로는 alpha 칩 혹은 mips 칩을 가진 환경에서 컴파일
될 경우 선언되게 됩니다.

4.4 xcoff_debug_hooks

이 정의는 $prefix/gcc/dbxout.c 파일에 정의되어 있습니다.

```
struct gcc_debug_hooks xcoff_debug_hooks =
{
  dbxout_init,
  dbxout_finish,
  debug_nothing_int_charstar,
  debug_nothing_int_charstar,
  dbxout_start_source_file,
  dbxout_end_source_file,
  xcoffout_begin_block,
  xcoffout_end_block,
  debug_true_tree,
  xcoffout_source_line,
  xcoffout_begin_prologue,
  debug_nothing_int,
  xcoffout_end_epilogue,
  debug_nothing_tree,
  xcoffout_end_function,
  debug_nothing_tree,
  dbxout_global_decl,
  debug_nothing_tree,
  debug_nothing_tree,
  debug_nothing_rtx
};
```

4.5 dwarf_debug_hooks

이 정의는 $prefix/gcc/dwarfout.c 파일에 정의되어 있습니다.

```
struct gcc_debug_hooks dwarf_debug_hooks =
{
  dwarfout_init,
  dwarfout_finish,
  dwarfout_define,
  dwarfout_undef,
  dwarfout_start_source_file_check,
  dwarfout_end_source_file_check,
  dwarfout_begin_block,
  dwarfout_end_block,
  debug_true_tree,
  dwarfout_source_line,
  dwarfout_source_line,
  dwarfout_end_prologue,
  dwarfout_end_epilogue,
  debug_nothing_tree,
  dwarfout_end_function,
  dwarfout_function_decl,
  dwarfout_global_decl,
  dwarfout_deferred_inline_function,
  debug_nothing_tree,
  debug_nothing_rtx
};
```

4.6 dwarf2_debug_hooks

이 정의는 $prefix/gcc/dwarf2out.c 파일에 정의되어 있습니다.

```
struct gcc_debug_hooks dwarf2_debug_hooks =
{
  dwarf2out_init,
  dwarf2out_finish,
  dwarf2out_define,
  dwarf2out_undef,
  dwarf2out_start_source_file,
  dwarf2out_end_source_file,
  dwarf2out_begin_block,
  dwarf2out_end_block,
  dwarf2out_ignore_block,
  dwarf2out_source_line,
  dwarf2out_begin_prologue,
  debug_nothing_int,
  dwarf2out_end_epilogue,
  debug_nothing_tree,
  debug_nothing_int,
  dwarf2out_decl,
  dwarf2out_global_decl,
  debug_nothing_tree,
  dwarf2out_abstract_function,
  debug_nothing_rtx
};
```

4.7 vmsdbg_debug_hooks

이 정의는 $prefix/vmsdbgout.c 파일에 선언되어 있습니다.

```
struct gcc_debug_hooks vmsdbg_debug_hooks =
{
  vmsdbgout_init,
  vmsdbgout_finish,
  vmsdbgout_define,
  vmsdbgout_undef,
  vmsdbgout_start_source_file,
  vmsdbgout_end_source_file,
  vmsdbgout_begin_block,
  vmsdbgout_end_block,
  vmsdbgout_ignore_block,
  vmsdbgout_source_line,
  vmsdbgout_begin_prologue,
  debug_nothing_int,
  vmsdbgout_end_epilogue,
  vmsdbgout_begin_function,
  debug_nothing_int,
  vmsdbgout_decl,
  vmsdbgout_global_decl,
  debug_nothing_tree,
  vmsdbgout_abstract_function,
```

```
  debug_nothing_rtx
};
```

제 5 절 그럼 GCC 내에서 언제 설정되나

GCC 가 실행이 될 경우 사용자는 GCC 를 위한 옵션을 command line 에 정의를 해주게 되는게, debug 관련 옵션은 모두 $prefix/gcc/toplev.c 파일에 선언되어 있는 decode_g_option () 함수에서 처리하게 됩니다.

그리고 이 함수내부에서 전역 열거자인 write_symbols 의 값을 변경하게 됩니다. 원래 기본값은 아래와 같습니다.

```
enum debug_info_type write_symbols = NO_DEBUG;
```

잠시 열거자 debug_info_type 에 대해서 말씀을 드리면 아래와 같이 선언되어 있는 열거자입니다. $prefix/gcc/flags.h 파일에 정의되어 있습니다.

```
enum debug_info_type
{
  NO_DEBUG,
  DBX_DEBUG,
  SDB_DEBUG,
  DWARF_DEBUG,
  DWARF2_DEBUG,
  XCOFF_DEBUG,
  VMS_DEBUG,
  VMS_AND_DWARF2_DEBUG
};
```

위에서 dbx, sdb 등등과 같은 단어들에 대한 설명을 많이 했으므로 하지 않겠습니다. 약간 특이한 것이 있다면 마지막 항목인 VMS_AND_DWARF2_DEBUG 에 관한 것인데 이것은 저도 정확하게 잘 모르지만 VMS debug info (vmsdbgout.c 를 사용) 와 DWARF v2 debug info 를 모두 작성 (dwarf2out.c 를 사용) 하는 것 처럼 보입니다.

그리고 한가지 더 짚고 넘어가야 할 부분이 있는데 그것은
enum debug_info_level 에 관한 것입니다. 이것은 debug info 에 대한 level 을 정의하는 것으로써 아래와 같이 선언되어 있습니다. 이것도 $prefix/gcc/flags.h 파일에 정의되어 있습니다.

```
enum debug_info_level
{
  DINFO_LEVEL_NONE,
  DINFO_LEVEL_TERSE,
  DINFO_LEVEL_NORMAL,
  DINFO_LEVEL_VERBOSE
};
```

각 항목에 대한 설명은 아래와 같습니다.

- DINFO_LEVEL_NONE
  Debugging info 를 작성하지 않습니다.
- DINFO_LEVEL_TERSE
  Traceback 들만 지원하게끔 최소한의 info 를 작성합니다.
- DINFO_LEVEL_NORMAL
  모든 선언들 (그리고 Line table) 를 위한 info 를 작성합니다.
- DINFO_LEVEL_VERBOSE
  보통의 info 에 #define/#undef info 도 포함합니다.

이것은 GCC 옵션에서 '-g¡level¿' 이런 식으로 숫자를 적어줄 경우 설정되게 됩니다. 이 열거자를 다룰 전역 열거자가 $prefix/gcc/toplev.c 파일에 선언되어 있으며 기본형으로 아래와 같이 선언됩니다.

```
enum debug_info_level debug_info_level = DINFO_LEVEL_NONE;
```

이것의 정의로써 우리가 생성하고자 하는 debugging information 의 level 를 알수 있습니다.

제 6 절 질문/답변 공간

음.. 이 공간은 이번 강의와 주제는 비슷하지만 따로 섹션을 만들어 내용을 보충하기에는 조금 주제가 작은 것들을 질답 형식으로 풀어내어 정보를 전달하고자 합니다.

[질문] 만약 GCC 를 컴파일할 때 GDB 를 위한 debug format 을 지정하기 위해서는 '-ggdb' 이러한 옵션을 따로 붙이게 되는게 이렇게 입력해 주면 GCC 내에서는 어떤 debug format 을 사용하게 되는 건가요?

[답변] '-ggdb' 옵션을 사용할 경우 GCC 는 DWARF version 2 debug format 을 선택하게 됩니다.

제 7 절 10 주차 강의를 마치며

다음 주 강의에는 그 동안 다른 것의 설명으로 인해 미루어졌던 "언어 독립적인 부분의 초기화"에 대한 강의를 하게 되는군요. 이 부분에 대한 강의를 작성하는데는 조금 많은 시간이 걸릴 것 같네요. 제목은 개요라지만 원체 제가 모르는 부분에 대한 것들이 많기 때문에 저도 공부를 해야 하고 자료 조사도 해야 좀 더 낳은 형태의 강의를 작성할 수 있을 것 같습니다. 이번 주의 경우는 많은 쓰임새가 있는 구조체를 살펴본 것은 아니지만 나중에 한번 더 다루어야 할 부분에 대한 서두 정도로 평가하고 싶습니다. 그럼 다음 강의에서 뵙겠습니다. 다음 주가 아니랍니다. :(