# CDB File Format

Lenny Story

2003-03-21

SDCC Development Team

## Contents

# 1 Overview

The CDB File is used to record all of the information that describes the variables, functions, lines, and memory items. These records provide the critical information that allows external utilities to properly locate and interpret variables, functions, and types. Development tools such as simulators debuggers and profilers use these records to analyze and describe the code in terms of the high level language in which it was written. The CDB file is therefore the critical link to bridging the compiled code image to the source files from which it was built.

# 2 Usage

CDB files are created when the "--debug" option is used. Each source file will have its own CDB file associated with it. When the --debug flag is used during the link process, the CDB file of the FIRST source module will contain all the records from all the source modules linked as well as the linker records.

Code and variables that are removed during the optimization phase will not be present in the debug file. Images created for debugging purposes should always disable as much optimization as possible. The following options are critical for the completeness of the resulting debugging file.

--debug Mandatory to produce a debugging file.

--noinduction Induction processing occurs at the very latest stages of compilation. This results in the inability to report the location of the resulting variables. Including this option disables the loop induction optimization.

--nooverlay Overlay variables will not show up in the debug file. Including this option will force variables to the data segment.

# 3 Conventions

The record examples and grammar shown in this document are displayed on multiple lines only for the purposes of readability. The records contained within the CDB files are always encoded on a single line.

Record grammar is indicated here using a custom format of the following specifications:

- Record Elements are surrounded using the characters ‘<’ and ’>’.
- Alternation is indicated using the ‘|’ character.
- Non-mandatory items are surrounded using the characters ‘{‘ and ‘}’.

# 4 Record Formats

## 4.1 Basic Record Format

**<RecordType><:><RecordSpec>**

| Type | Description | Form |
|---|---|---|
| M | Module Record | Compiler |
| F | Function Record | Compiler |
| S | Symbol Record | Compiler |
| T | Structure (Complex Type) Record | Compiler |
| L | Linker Record | Assembler / Linker |

All records are ASCII text, with one record per line. The record type is the first character, followed by a single colon ":". Sub types are often included as part of the record specific format.

## 4.2 Module Record

**<M><:><Filename>**

| | |
|---|---|
| M | Module record type indicator |
| Filename | The filename of the module that this CDB file represents. |

**Purpose**

The module record is used to define a source module. It is usually used at the beginning of each of the module specific CDB files, and will occur several times in the final CDB file produced by the linker to represent each of the source modules. The location of this record within the file does not necessarily indicate the scope of the variables that follow. Those variables that have module specific scope have an indication encoded within their respective symbol entry.

**Examples**

```
M:Timer0
M:_bp
```

## 4.3 Symbol Records

| | |
|---|---|
| S | Symbol record type indicator |
| G | Scope is global |
| F <Filename> | Scope is file |
| L <Function> | Scope is local |
| <Name> | Symbol name |
| <Level> | Scope level (see below) |
| <Block> | Scope block (see below) |
| <TypeChain> | Type chain record (see type record below) |
| <Address Space> | Address space code (see table below) |
| <On Stack> | Is this symbol on the stack? Indicates if the next parameter is valid. |
| <Stack> | The stack Offsetorelative to the “bp” variable. (The libraries may have to be recompiled to include the --debug option for this variable to appear in the debug file). |
| <Reg> | If the address space is ’R’, this field indicates the register that the symbol is allocated. The register name is in its native form: R1, AX, etc. |

**<S><:>**
**{ G | F<Filename> | L { <function> | “–null–“ }}**
**<$><Name>**
**<$><Level>**
**<$><Block>**
**<(><TypeRecord><)>**
**<,><AddressSpace>**
**<,><OnStack>**
**<,><Stack>**
**<,><[><Reg><,>{<Reg><,>}<]>**

A symbol record is generated for each named symbol in the source file; this includes local, global and parameter symbols.

**Blocks and Levels**

The level & block are used to further scope local variables since C allows unique definitions across different scope blocks. When using the symbol records, it is always important to include the level and block information as part of the identification. It is possible to have two symbols that share the same name, but have different scope information.

The linker address records contain not only the name of the symbol, but the Scope information as well, which is instrumental in determining the correct instantiation of the symbol.

Linker C line records also contain the Scope information (see below).

The following code fragment illustrates a simple scope example:

```
foo()
{
int c; /* block #1 , level #1 */
   {
      int c; /* block #2, level #2 */
      ...
   }
   {
      int c; /* block #3 , level #2 */
      ...
   }
}
```

| | |
|---|---|
| A | External stack |
| B | Internal stack |
| C | Code |
| D | Code / static segment |
| E | Internal ram (lower 128) bytes |
| F | External ram |
| G | Internal ram |
| H | Bit addressable |
| I | SFR space |
| J | SBIT space |
| R | Register space |
| Z | Used for function records, or any undefined space code |

## 4.4 Type Chain Record

**<{><Size><}> <DCLType> <,> {<DCLType> <,>} <:> <Sign>**

| | |
|---|---|
| <Size> | The size of the item in decimal. |
| <DCLType> | The type encoded using the table below. |
| <Sign> | The sign of the item. Encoded as 'U' or 'S'. |

**Purpose**

The C programming language allows arbitrarily complex type constructions. Because of this, the CDB file type designations are organized as a list of basic primitive types.

| | |
|---|---|
| DA <n> | Array of n elements |
| DF | Function |
| DG | Generic pointer |
| DC | Code pointer |
| DX | External ram pointer |
| DD | Internal ram pointer |
| DP | Paged pointer |
| DI | Upper 128 byte pointer |
| SL | long |
| SI | int |
| SC | char |
| SS | short |
| SV | void |
| SF | float |
| ST <name> | Structure of name <name> |
| SX | sbit |
| SB <n> | Bit field of <n> bits |

**Examples**

```
S:LcheckSerialPort$pstBuffer$1$1({3}DG,STTTinyBuffer:S),R,0,0,[r2,r3,r4]
S:Ltimer0LoadExtended$count$1$1({2}SI:S),B,1,-4
S:G$T2CON_7$0$0({1}SX:S),J,0,0
S:LAdcInitialize$a$1$1({2}SI:S),B,1,1
S:G$ScanCount$0$0({2}SI:S),F,0,0
```

## 4.5 Function Records

**<F><:>**

**{ G | F<Filename> | L { <function> | "–null–" }}**
**<$><Name>**
**<$><Level>**
**<$><Block>**
**<(><TypeRecord><)>**
**<,><AddressSpace>**
**<,><OnStack>**
**<,><Stack>**
**<,><Interrupt>**
**<,><Interrupt Num>**
**<,><Register Bank>**

| | |
|---|---|
| F | Symbol record type indicator |
| G | Scope is global |
| F <Filename> | Scope is file |
| L <Function> | Scope is local |
| <Name> | Symbol name |
| <Level> | Scope level (see below) |
| <Block> | Scope block (see below) |
| <TypeChain> | Type chain record (see type record below) |
| <Address Space> | Address space code <see table below> |
| <On Stack> | Indicates if this is a stack variable |
| <Stack> | If stack variable, the stack offset relative to the "bp" variable. (Libraries will have to be compiled using the --debug option for this to be available) |
| <Is Interrupt> | Indicates if this is an interrupt handler. |
| <Interrupt Num> | If interrupt handler, this indicates the interrupt number. |
| <Register Bank> | If interrupt handler, this ist he register bank number. |

**Purpose**

The Function record defines any Source File function. Its construction is the same as the symbol record, with the addition of 3 extra parameters for indicating interrupt handlers.

**Example**

```
F:G$main$0$0({2}DF,SV:S),C,0,0,0,0,0
F:G$SioISR$0$0({2}DF,SV:S),Z,0,0,1,4,0
```

## 4.6 Type Records

**<T><:>**
**<F><Filename><$>**
**<Name>**
**<[><TypeMember> {<TypeMember>} <]>**

| | |
|---|---|
| T | Type record type indicator |
| <Filename> | The filename where this type is declared |
| <Name> | The name of this type |
| <TypeMember> | (see below) |

## 4.7 Type Member

**<(><{><Offset><}><SymbolRecord><)>**

| | |
|---|---|
| <Offset> | The offset of this type member in decimal. |
| <SymbolRecord> | A complete symbol record describing this Member. (See "Symbol Records" above. |

**Purpose**

Type records describe the complex types within the source file. These include structure and union types.

**Examples**

```
T:Fcmdas$TTinyBuffer[
({0}S:S$pNext$0$0({3}DG,STTTinyBuffer:S),Z,0,0)
({3}S:S$length$0$0({1}SC:U),Z,0,0)
({4}S:S$maxLength$0$0({1}SC:U),Z,0,0)
({5}S:S$rindex$0$0({1}SC:U),Z,0,0)
({6}S:S$windex$0$0({1}SC:U),Z,0,0)
({7}S:S$buffer$0$0({64}DA64,SC:U),Z,0,0)
]
```

## 4.8 Link Address of Symbol

**<L><:>**
**{ <G> | F<filename> | L<function> }**
**<$><name>**
**<$><level>**
**<$><block>**
**<:><address>**

| | |
|---|---|
| L | Link record type indicator |
| G | Symbol has file scope. |
| F <Filename> | Symbol has file scope. |
| L <Function> | Symbol has function scope |
| <Name> | Symbol name |
| <Level> | Symbol level |
| <Block> | Symbol block |
| <Address> | Symbol address in hex, relative to the address space code, in the matching symbol record. |

**Purpose**

The link address record is used to bind a memory location to a symbol record.

**Example**

```
L:G$P0$0$0:80
L:G$ScanCount$0$0:0
L:Fcmdas$_str_0$0$0:195
```

## 4.9 Linker Symbol End Address Record

**<L><:><X>**
**{ <G> | F<filename> | L<functionName> }**

**<$><name>**
**<$><level>**
**<$><block>**
**<:><Address>**

| | |
|---|---|
| L | Link record type indicator |
| X | Link end address sub type indicator |
| G | Symbol has file scope. |
| F <Filename> | Symbol has file scope. |
| L <Function> | Symbol has function scope. |
| <Name> | Symbol name |
| <Level> | Symbol level |
| <Block> | Symbol block |
| <Address> | Symbol end address in hex, relative to the address space code contained in the matching symbol record. |

**Purpose**

The Linker Symbol end address record is primarily used to indicate the Ending address of functions. This is because function records do not contain a size value, as symbol records do.

**Example**

```
L:XG$sysClearError$0$0:194
L:XG$SioISR$0$0:A09
```

## 4.10 Linker ASM Line Record

**<L> <:> <A>**
**<$><Filename>**
**<$><Line>**
**<:><EndAddress>**

| | |
|---|---|
| L | Link record type indicator |
| A | Link assembly file line record sub type indicator |
| <Filename> | Filename of the assembly file. |
| <Line> | Line number in the above filename.{}These numbers start at 1 (not 0). |
| <EndAdress> | End address |

**Purpose**

The linker Asm Line record is used to bind the execution address with a source file and line number.

**Example**

```
L:A$TinyBuffer$2320:A13
L:A$max1270$391:CA4
```

## 4.11 Linker C-Line Record

**<L> <:> <C>**
**<$><Filename>**
**<$><Line>**
**<$><Level>**
**<$><Block>**
**<:><EndAddress>**

| | |
|---|---|
| L | Link record type indicator |
| C | Link assembly file line record sub type indicator |
| <Filename> | Filename of the assembly file. |
| <Line> | Line number in the above filename. These numbers start at 1 (not 0). |
| <Level> | Current level at this line and address. |
| <Block> | Current block at this line and address. |
| <EndAddress> | End address |

**Purpose**

The linker C-Line record is used to bind the execution address with a source file, line number and the level, block information.

The following is an example source module with its lines showing the corresponding C-Line Records. You will notice in this example that there are no line entries for lines 18, 19. This is because the code was optimized and removed.

```
01
02                      struct complex
03                      {
04                        int count;
05                        int Max;
06                      };
07
08  L:C$vars.c$8$0$0:38 void main(void)
09                      {
10                        int iterA;
11                        int iterB;
12                        struct complex myStruct;
13
14  L:C$vars.c$14$1$1:C1    for(iterA = 0; iterA < 10; iterA++)
15                          {
16  L:C$vars.c$16$2$2:B9      for(iterB = 0; iterB < 10; iterB++)
17                            {
18                              int iterA = 6 + iterB;
19                              iterA++;
20
21  L:C$vars.c$21$1$1:69        myStruct.count++;
22
23  L:C$vars.c$23$3$3:83        if(myStruct.count > myStruct.Max)
24  L:C$vars.c$24$3$3:A3          myStruct.Max = mystruct.count;
25                            }
26                          }
27  L:C$vars.c$27$1$1:CE  }
```

**Example**

```
L:C$max1270.c$35$1$1:CA9
L:C$Timer0.c$20$1$1:D9D
```

# 5 Source File Example

```
sfr IM = 0x90;
struct complex
{
  int count;
  int Max;
};
void main(void)
{
  int iterA;
  int iterB;
  struct complex myStruct;
```

```
      for(iterA = 0; iterA < 10; iterA++)
      {
        for(iterB = 0; iterB < 10; iterB++)
        {
          int iterA = 6 + iterB;
          iterA++;
          myStruct.count++;

          IM = iterA;
          if(myStruct.count > myStruct.Max)
            myStruct.Max = myStruct.count;
        }
      }
    }
```

# 6 CDB File Example

```
M:vars
F:G$main$0$0({2}DF,SV:S),C,0,0,0,0,0
T:Fvars$complex[({0}S:S$count$0$0({2}SI:S),Z,0,0)({2}S:S$Max$0$0({2}SI:S),Z,0,0)
S:Lmain$iterA$1$1({2}SI:S),R,0,0,[r0,r1]
S:Lmain$iterB$1$1({2}SI:S),R,0,0,[r4,r5]
S:Lmain$myStruct$1$1({4}STcomplex:S),E,0,0
S:Lmain$iterA$3$3({2}SI:S),R,0,0,[r6,r7]
S:G$IM$0$0({1}SC:U),I,0,0
S:G$main$0$0({2}DF,SV:S),C,0,0
L:G$IM$0$0:90
L:Lmain$myStruct$1$1:8
L:A$vars$64:0
L:A$vars$65:3
L:A$vars$67:B
L:A$vars$69:13
L:A$vars$71:1B
L:A$vars$73:23
L:A$vars$75:2B
L:A$vars$129:33
L:A$vars$131:36
L:A$vars$158:38
L:C$vars.c$10$0$0:38
L:G$main$0$0:38
L:A$vars$159:3A
L:A$vars$163:3C
L:A$vars$164:3D
L:A$vars$165:3E
```

```
L:A$vars$166:40
L:A$vars$167:41
L:A$vars$168:43
L:A$vars$170:45
L:A$vars$171:47
L:A$vars$176:4A
L:A$vars$177:4C
L:A$vars$181:4E
L:A$vars$182:4F
L:A$vars$183:50
L:A$vars$184:52
L:A$vars$185:53
L:A$vars$186:55
L:A$vars$191:57
L:A$vars$196:59
L:C$vars.c$20$3$3:59
L:A$vars$197:5B
L:A$vars$198:5D
L:A$vars$200:5E
L:A$vars$201:5F
L:A$vars$202:61
L:A$vars$207:62
L:C$vars.c$21$3$3:62
L:A$vars$208:63
L:A$vars$209:66
L:A$vars$216:67
L:C$vars.c$23$3$3:67
L:A$vars$217:69
L:A$vars$220:6B
L:A$vars$221:6C
L:A$vars$222:6F
L:A$vars$227:70
L:A$vars$228:72
L:A$vars$232:74
L:C$vars.c$25$3$3:74
L:A$vars$238:76
L:C$vars.c$27$3$3:76
L:A$vars$239:77
L:A$vars$240:79
L:A$vars$241:7A
L:A$vars$242:7C
L:A$vars$243:7E
L:A$vars$244:80
L:A$vars$245:83
L:A$vars$248:85
L:A$vars$256:87
```

```
L:C$vars.c$28$3$3:87
L:A$vars$257:8A
L:A$vars$263:8D
L:C$vars.c$18$2$2:8D
L:A$vars$264:8E
L:A$vars$265:91
L:A$vars$268:92
L:A$vars$274:94
L:C$vars.c$16$1$1:94
L:A$vars$275:95
L:A$vars$276:98
L:A$vars$278:99
L:A$vars$282:9C
L:C$vars.c$31$1$1:9C
L:XG$main$0$0:9C
L:A$vars$84:A1
L:A$vars$85:A4
L:A$vars$86:A7
L:A$vars$87:A9
L:A$vars$88:AB
L:A$vars$91:AE
L:A$vars$92:B0
L:A$vars$93:B2
L:A$vars$94:B4
L:A$vars$95:B6
L:A$vars$96:B8
L:A$vars$97:B9
L:A$vars$98:BB
L:A$vars$99:BD
L:A$vars$100:BE
L:A$vars$101:C1
L:A$vars$102:C3
L:A$vars$103:C6
L:A$vars$104:C7
L:A$vars$105:C8
L:A$vars$106:C9
L:A$vars$107:CA
L:A$vars$108:CB
L:A$vars$109:CE
L:A$vars$110:D0
L:A$vars$111:D2
L:A$vars$112:D5
L:A$vars$113:D7
L:A$vars$114:DA
L:A$vars$118:DD
```