Open Watcom Linux Port
Compiler / Linker Software Requirements Specification

Created on 1/14/2004 11:08 AM

Table of Contents

Compiler / Linker Software Requirements Specification

Executive summary

This document describes a detailed approach to porting Open Watcom Compiler and Linker to Linux platform.

1. Introduction

This document outlines a set of steps that should be taken to provide shared libraries and position independent code support to the Open Watcom compiler as a part of the Open Watcom Linux porting effort.

All information is presented relative to **open_watcom_devel_1.1.7**. Since Open Watcom is open-source project, we assume some of the topics covered might become obsolete or inaccurate at the moment of reading this document. A considerable amount of experimental work was performed prior writing this Specification. Some results of that work are included in this document.

This document consists of four large sections. Section one is an introductory section. Section two describes the key components of Open Watcom C Compiler and Linker. Section three defines steps, needed for adding PIC and shared object support. Section four describes some problems found during our investigation.

1.1 Definitions, acronyms and abbreviations

ABI	**A**pplication **B**inary **I**nterface
ELF	**E**xecutable and **L**inking **F**ormat There are three main types of ELF files: ▪ A relocatable file holds code and data suitable for linking with other object files to create an executable or a shared object file. ▪ An executable file holds a program suitable for execution; the file specifies how the function exec() creates a program's process image. ▪ A shared object file holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image. wlink have limited support of building and using of ELF files.
OMF	Relocatable **O**bject **M**odule **F**ormat This format (developed by Microsoft) is produced by wcc386 (and has "native" support in wlink).
ORL	**O**bject **R**eading **L**ibrary API for reading object files.
PDC	**P**osition **D**ependent **C**ode (opposite to PIC)
PIC	**P**osition **I**ndependent **C**ode This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Because PIC uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The difference between the virtual address of any segment in memory and the corresponding virtual address in the file is thus a single constant value for any one executable or shared object in a given process.
wcc386	Open Watcom C Compiler
wlink	Open Watcom Linker

1.2 References

1. SYSTEM V APPLICATION BINARY INTERFACE, Edition 4.1
2. SYSTEM V APPLICATION BINARY INTERFACE, Intel386™ Architecture Processor Supplement, Fourth Edition
3. Linux Standard Base Specification for the IA32 Architecture 1.9.0-20031030
4. OMF 1.1 Specification

2. Key Components of the Open Watcom C Compiler and Linker

Certain parts of the Open Watcom source code are especially important for our project. Such parts will be referred as "components" throughout this document, although some of them are logically interrelated source files, and others are subprojects (subdirectories under the Open Watcom source tree). The informal names defined here will be used in the further parts of this document.

Each component is described in two sections. First section describes the purpose of the component, and provides the list of core source files. Second section describes the principles of function of the corresponding component. Important functions, data structures, and constants are described as well.

2.1 ORL

2.1.1 Definition

Abbreviation of "Object Reading Library". Located in **$OWROOT/bld/orl**.

ORL is designed for reading various formats of object files: ELF, OMF, and COFF. We are interested mainly in the ELF stuff (**$OWROOT/bld/orl/elf**).

ELF linking information (e.g. relocation entries) is mapped to abstract ORL linking information. For example, ELF relocation type **R_386_32** is mapped to **ORL_RELOC_TYPE_WORD_32**.

2.1.2 Description

There are several handle types defined in ORL. Most important are **orl_sec_handle** and **orl_symbol_handle**. There are many functions operating with sections and symbols:

```
char *            ORLSecGetName( orl_sec_handle );
orl_sec_offset    ORLSecGetBase( orl_sec_handle );
orl_sec_size      ORLSecGetSize( orl_sec_handle );
orl_sec_type      ORLSecGetType( orl_sec_handle );
orl_sec_flags     ORLSecGetFlags( orl_sec_handle );
orl_sec_alignment ORLSecGetAlignment( orl_sec_handle );
orl_sec_handle    ORLSecGetStringTable( orl_sec_handle );
orl_sec_handle    ORLSecGetSymbolTable( orl_sec_handle );
orl_sec_handle    ORLSecGetRelocTable( orl_sec_handle );
orl_linnum *      ORLSecGetLines( orl_sec_handle );
orl_table_index   ORLSecGetNumLines( orl_sec_handle );
orl_sec_offset    ORLSecGetOffset( orl_sec_handle );
orl_return        ORLSecGetContents( orl_sec_handle, char ** );
```

```
orl_return        ORLSecQueryReloc( orl_sec_handle, orl_sec_offset,
orl_reloc_return_func );
orl_return        ORLSecScanReloc( orl_sec_handle, orl_reloc_return_func );
orl_table_index   ORLCvtSecHdlToIdx( orl_sec_handle );
orl_sec_handle    ORLCvtIdxToSecHdl( orl_file_handle, orl_table_index );
char *            ORLSecGetClassName( orl_sec_handle );
orl_sec_combine   ORLSecGetCombine( orl_sec_handle );
orl_sec_frame     ORLSecGetAbsFrame( orl_sec_handle );
orl_sec_handle    ORLSecGetAssociated( orl_sec_handle );
orl_group_handle  ORLSecGetGroup( orl_sec_handle );
orl_return        ORLRelocSecScan( orl_sec_handle, orl_reloc_return_func );
orl_return        ORLSymbolSecScan( orl_sec_handle, orl_symbol_return_func );
orl_return        ORLNoteSecScan( orl_sec_handle, orl_note_callbacks *, void *
);


char *             ORLSymbolGetName( orl_symbol_handle );
orl_symbol_value   ORLSymbolGetValue( orl_symbol_handle );
orl_symbol_binding ORLSymbolGetBinding( orl_symbol_handle );
orl_symbol_type    ORLSymbolGetType( orl_symbol_handle );
unsigned char      ORLSymbolGetRawInfo( orl_symbol_handle );
orl_sec_handle     ORLSymbolGetSecHandle( orl_symbol_handle );
orl_symbol_handle  ORLSymbolGetAssociated( orl_symbol_handle );
```

These and other functions allow access to the object file in the uniform way. Actual mapping from ELF to ORL is performed by **$OWROOT/bld/orl/elf/c/elfentr.c** (sections, symbols), **elfload.c** (sections), **elflwlv.c** (symbols, relocations).

ORL is used by Open Watcom Linker, mostly in **$OWROOT/bld/wl/c/objorl.c**.

ELF relocations (i.e. 386 ABI) are mapped to abstract ORL relocations in the following way:

```
$OWROOT/bld/orl/elf/c/elflwlv.c
static orl_reloc_type convert386Reloc( elf_reloc_type elf_type ) {
    switch( elf_type ) {
    case R_386_NONE:
        return( ORL_RELOC_TYPE_ABSOLUTE );
    case R_386_32:
    case R_386_GOT32:
    case R_386_GOTOFF:
        return( ORL_RELOC_TYPE_WORD_32 );
```

```
        case R_386_PC32:
        case R_386_PLT32:
        case R_386_GOTPC:
            return( ORL_RELOC_TYPE_REL_32 );
        default:
            assert( 0 );
        }
        return( ORL_RELOC_TYPE_NONE );
    }
```

2.2 WLCore

2.2.1 Definition

Synthetically selected part of the Open Watcom Linker (**$OWROOT/bld/wl**), performing the basic linking tasks (e.g. relocations), and interacting to ORL. Main files: **obj2supp.c**, **objcalc.c**, **objorl.c**, **objpass1.c**, **objpass2.c**.

2.2.2 Description

We are interested mainly in ELF linking. Since ORL is used to read ELF object files, there is an interface to ORL implemented in **objorl.c**. The linker uses other data structures than ORL, so there is another mapping implemented in the mentioned file. Relocations are mapped in the following way:

```
switch( reloc->type ) {
// ...
case ORL_RELOC_TYPE_ABSOLUTE:
    type = FIX_OFFSET_32 | FIX_ABS;
    break;
// ...
case ORL_RELOC_TYPE_REL_32:
    type = FIX_OFFSET_32 | FIX_REL;
    break;
// ...
case ORL_RELOC_TYPE_WORD_32:
    type = FIX_OFFSET_32;
    break;
}
```

Constants **FIX_** are defined in the spirit of OMF specification (i.e. **FIXUPP** records). However, some of these constants implement specific features, e.g. PowerPC relocations.

```
$OWROOT/bld/wl/h/obj2supp.h

typedef enum {
    FIX_CHANGE_SEG      = 0x00000001,   // has to be 1.  used in pointers!
    FIX_ADDEND_ZERO     = 0x00000002,
    FIX_UNSAFE          = 0x00000004,
    FIX_ABS             = 0x00000008,

    FIX_BASE            = 0x00000010,
    FIX_HIGH            = 0x00000020,
    FIX_REL             = 0x00000040,
    FIX_SHIFT           = 0x00000080,

    FIX_TARGET_SHIFT    = 8,            // contains frame_type
    FIX_TARGET_MASK     = 0x00000700,

    FIX_NO_BASE         = 0x00001000,
    FIX_SIGNED          = 0x00002000,
    FIX_LOADER_RES      = 0x00004000,
    FIX_SEC_REL         = 0x00008000,

    FIX_NO_OFFSET       = 0,
    FIX_OFFSET_8        = 0x00010000,
    FIX_OFFSET_16       = 0x00020000,
    FIX_OFFSET_21       = 0x00030000,
    FIX_OFFSET_32       = 0x00040000,
    FIX_OFFSET_24       = 0x00050000,
    FIX_OFFSET_SHIFT    = 16,
    FIX_OFFSET_MASK     = 0x00070000,

    FIX_TOC             = 0x00100000,   // PPC PE
    FIX_TOCV            = 0x00200000,   // PPC PE
    FIX_IFGLUE          = 0x00300000,   // PPC PE
    FIX_SPECIAL_MASK    = 0x00300000,

    FIX_FRAME_SHIFT     = 24,           // contains frame_type
    FIX_FRAME_MASK      = 0x07000000,
```

```
// now for some handy constants which use these

    FIX_BASE_OFFSET_16  = (FIX_BASE | FIX_OFFSET_16),
    FIX_BASE_OFFSET_32  = (FIX_BASE | FIX_OFFSET_32),
    FIX_HIGH_OFFSET_8   = (FIX_HIGH | FIX_OFFSET_8),
    FIX_HIGH_OFFSET_16  = (FIX_HIGH | FIX_OFFSET_16),
} fix_type;
```

During the first pass, relocations are converted to internal representation:

```
$OWROOT/bld/wl/c/objorl.c
static orl_return P1Relocs( orl_sec_handle sec )
/*******************************************/
{
    return ORLRelocSecScan( sec, DoReloc );
}
```

Here **ORLRelocSecScan** is ORL-function that iterates through the relocation list, and **DoReloc()** is called to convert each relocation (see above).

Relocation processing is actually implemented in **obj2supp.c**. This file is a key part of the linker.

Other important participants of linking process are symbols. Like relocations, ELF symbols (accessible through ORL functions) are converted to the internal **symbol** structures:

```
$OWROOT/bld/wl/h/syms.h
typedef struct symbol {
    struct symbol *     hash;
    struct symbol *     publink;
    struct symbol *     link;
    targ_addr           addr;
    unsigned_16         namelen;
    sym_info            info;       // flags & floating point fixup type.
    struct mod_entry *  mod;
    union {
        void *          edges;      // for dead code elim. when sym undefd
        struct segdata *seg;        // seg symbol is in.
        char *          alias;      // for aliased syms.
        void *          import;     // NOVELL & OS/2 only: imported symbol
data.
```

```
        offset          cdefsize;   // altdef comdefs: size of comdef
    } p;
    union {
        dos_sym_data    d;
        struct symbol * altdefs;    // for keeping track of comdat & comdef
defs
        struct symbol * datasym;    // altdef comdats: sym which has data def
        int             aliaslen;   // for aliases - length of name.
    } u;
    union {
        struct symbol * mainsym;    // altdefs: main symbol definition
        struct symbol * def;        // for lazy externs
        struct symbol **vfdata;     // for virtual function lazy externs.
        void *          export;     // OS/2 & PE only: exported sym info.
    } e;
    char *              name;
    char *              prefix;     // primarily for netware, though could be
                                    // subverted for other use. gives symbol
                                    // namespace qualification
} symbol;
```

There are many **SYM_** and **ST_** constants describing various symbol properties.

Finally, calculation of segment addresses (during the second pass) is performed in **objcalc.c**. Information produced during this process will be used later for creating an executable file. One can iterate through the groups (i.e. grouped segments) this way:

```
group_entry *currgrp;
for( currgrp = Groups; currgrp != NULL; currgrp = currgrp->next_group ) {
    // Do something...
}
```

There are some important global variables. In the example above, we see **Groups** is the list of all groups. Variable **DataGroup** specifies the data group, variable **NumGroups** contains the total number of groups. Group entry is defined as:

```
$OWROOT/bld/wl/h/objstruc.h
typedef struct group_entry {
    GROUP_ENTRY *       next_group;
    SEG_LEADER *        leaders;
```

```
    symbol *            sym;
    section *           section;
    targ_addr           grp_addr;
    unsigned_16         segflags;
    offset              size;
    offset              totalsize;
    offset              linear;         // preferred base address
    union {
        void *          grp_relocs;     // OS2/ELF only.
        class_entry *   class;          // CV (during addr calc )
    } g;
    union {
        unsigned        qnxflags;       // QNX
        unsigned        miscflags;      // OS/2
    } u;
    unsigned            num;
    unsigned            isfree : 1;
    unsigned            isautogrp : 1;
} group_entry;
```

Here **size** is group size in the file; **totalsize** is group size in the memory (e.g. uninitialized data do not require space in the file).

Another important global variable is **FmtData**. This structure contains fields describing the format and various properties of the output file. For our purposes, the most important fields are **type** and **dll**. For ELF shared objects, the following test evaluates as **TRUE**: **(FmtData.type & MK_ELF) && FmtData.dll**.

2.3 Load ELF

2.3.1 Definition

Part of Open Watcom Linker (**$OWROOT/bld/wl**), designed for writing executable files in ELF format. Consists of two files: **loadelf.c** and **loadelf2.c**.

2.3.2 Description

Currently, LoadELF is able to create only ELF executable files (shared objects are not supported). Most of the work is performed in **loadelf.c**. The second file, **loadelf2.c**, contains only the routines for creating ELF symbol tables.

The main function is **FiniELFLoadFile()**. The following tasks are performed there:

1. Initialize the ELF header, program headers, and section headers.
2. Write groups (i.e. code and data) to the ELF file (program and section headers are changed during this process; i.e. sections: **.text**, **.data**, and **.bss**).
3. Write relocation section (**.rela.text**).
4. Write DWARF debug information (if needed).
5. Write symbol table (**.symtab**), hash (**.hash**), and strings (**.strtab**).
6. Write section strings (**.shstrtab**).
7. Write section headers.
8. Write DWARF trailer (if needed).
9. Rewind and write the ELF header and program headers.

Task 1 is performed in **void SetHeaders(ElfHdr *hdr)**. Sections are initialized in **void InitSections(ElfHdr *hdr)**.

ElfHdr is defined as:

```
$OWROOT/bld/wl/h/loadelf2.h

typedef struct {
    Elf32_Ehdr eh;
    Elf32_Shdr *strhdr;
    Elf32_Phdr *ph;
    unsigned   ph_size;
    Elf32_Shdr *sh;
    unsigned   sh_size;
    stringtable secstrtab;
    struct {
        int        secstr;  // Index of strings section for section names
        int        grpbase; // Index base for Groups in section
        int        grpnum;  // Number of groups
        int        relbase; // Index base for relocation sections
```

```
        int         relnum;  // number of relocations
        int         symstr;  // Index of symbol's string table
        int         symtab;  // Index of symbol table
        int         symhash; // Index of symbol hash table
        int         dbgbegin;// Index of first debug section
        int         dbgnum;  // Number of debug sections
    } i;  // Indexes into sh
    unsigned_32 curr_off;
} ElfHdr;
```

The most interesting structure is **i**, where section indexes are specified. This structure is filled in **InitSections()**. So the order of sections is predefined.

Program header is created in **SetHeaders()** as well.

Task 2 is performed in **void WriteELFGroups(ElfHdr *hdr)**. In this function, group list is iterated (as described in WLCore). For each group, code or data are written to the ELF file, using **WriteGroupLoad()**. The corresponding program headers and sections are filled as well, using **SetGroupHeaders()**. Note that uninitialized data (**.bss**) are processed in the special way.

Relocations are written using **void WriteRelocsSections(ElfHdr *hdr)**. In this implementation, all relocations are presented with explicit addends (i.e. **SHT_RELA**).

Task 5 is performed by **WriteElfSymTable(ElfSymTable *tab, ElfHdr *hdr, int hashidx, int symtabidx, int strtabidx)**. Both symbol table and hash are written in this function. Then string table is written using **WriteSHStrings()**.

Function **WriteSHStrings()** is reused for the next task (i.e. writing section names).

Note that field **curr_off** (from **ElfHdr**) is widely used. This field specifies the current offset in the ELF file. However, it is not updated automatically, e.g. after **WriteLoad()** therefore precise calculations are needed to keep this value up to date.

Functions to write ELF (and other) executable files are located in **$OWROOT/bld/wl/c/loadfile.c**.

2.4 GC386

2.4.1 Definition

Code Generator for 32-bit family of x86 CPUs, used by Open Watcom C Compiler consists of three “layers”:

- General Code Generator, located in **$OWROOT/bld/cg**.
- Common x86 Code Generator (16/32-bit), located in **$OWROOT/bld/cg/intel**.
- Specific 32-bit x86 Code Generator, located in **$OWROOT/bld/cg/intel/386**.

2.4.2 Description

Documentation for the Code Generator (**$OWROOT/bld/cg/doc**) covers only the interface to the code generator (i.e. "General Code Generator"). The code generator (back end) interface is a set of procedure calls. These are divided into Code Generation (**CG**), Data Generation (**DG**), miscellaneous Back End (**BE**), Front end supplied (**FE**), and debugger information (**DB**) routines.

There is internal machine-independent format (allowing scalability, multiple platforms, and machine-independent optimizations). The main parts of these intermediate data (passed to the code generator for particular machine) are "blocks" and "instructions":

```
$OWROOT/bld/cg/h/block.h

typedef struct block {
        struct block_ins        ins;
        struct block            *next_block;    /* used for DFS */
        struct block            *prev_block;
        union {
            struct interval_def *interval;
            struct block        *partition;
            struct block        *loop;
        } u;
        struct block            *loop_head;
        struct data_flow_def    *dataflow;
        struct block_edge       *input_edges;
        pointer                 cc;             /* AKA cc_control */
        dominator_info          dom;            /* least node in dominator set
*/
        type_length             stack_depth;    /* set by FlowSave stuff */
        union {
            struct block        *alter_ego;     /* used in loop unrolling */
            struct block        *next;          /* used for CALL_LABEL kludge
*/
        } v;
        label_handle            label;          /* front end identification */
        local_bit_set           available_bit;
        interval_depth          depth;          /* loop nesting depth */
        block_num               id;             /* internal identification */
        block_num               gen_id;
        block_num               inputs;         /* number of input edges */
        block_num               targets;        /* number of target blocks */
        block_class             class;
        signed_32               iterations;
        unsigned_32             unroll_count;
        struct block_edge       edge[ 1 ];
```

```
} block;
$OWROOT/bld/cg/h/inslist.h
typedef struct ins_header {
        struct instruction      *prev;
        struct instruction      *next;
        struct name_set         live;
        source_line_number      line_num;
        opcode_defs             opcode;
        instruction_state       state;
} ins_header;

typedef struct instruction {
        struct ins_header       head;
        struct opcode_entry     *table;
        union {
            struct opcode_entry *gen_table;
            struct instruction  *parm_list;
            struct instruction  *cse_link;
        } u;
        struct register_name    *zap;
        union name              *result;        /*  result location */
        instruction_id          id;
        type_class_def          type_class;
        type_class_def          base_type_class;
        unsigned_16             sequence;
#include "cgnoalgn.h"
        union {
                byte            byte;
                bool            bool;
                call_flags      call_flags;
                nop_flags       nop_flags;
                byte            zap_value;      /* for conversions on AXP */
        }                       flags;
        union {
            byte                index_needs;    /*  a.k.a. reg_set_index */
            byte                stk_max;
```

```
        } t;
        byte                    stk_entry;
        byte                    num_operands;
        instruction_flags       ins_flags;
        byte                    stk_exit;
        union {
            byte                stk_extra;
            byte                stk_depth;
        }                       s;
#include "cgrealgn.h"
        union name              *operands[ 1 ]; /*  operands */
} instruction;
```

Sample: Walking through the blocks and instructions

```
block       *blk;
instruction *ins;

blk = HeadBlock;
while( blk != NULL ) {
    ins = blk->ins.hd.next;
    while( ins->head.opcode != OP_BLOCK ) {
        // Do something...
        ins = ins->head.next;
    }
    blk = blk->next_block;
}
```

Instructions are machine-independent. For example, **opcode == OP_ADD** specifies addition. Operands and result have the **name** type that can represent CPU register, memory location, immediate constant, etc.:

```
$OWROOT/bld/cg/h/name.h
typedef union name {
        struct  name_def        n;
        struct  var_name        v;
        struct  const_name      c;
        struct  memory_name     m;
```

```
        struct  temp_name       t;
        struct  register_name   r;
        struct  indexed_name    i;
        union   name            *_n;
} name;
```

This data looks like machine-dependent, since different architectures have different registers. However, this is top-level abstraction. The code generator for particular architecture supplies the corresponding set of registers. Actually there are many register sets (e.g. stack pointer, registers for temporary storage, fixed registers, etc.) The **hw_reg_set** type is able to hold one or more registers (or be empty).

For 32-bit family of x86 processors, the register sets are defined in **$OWROOT/cg/intel/386/c/386rgtbl.c**.

There are many inline functions operating with register sets (**$OWROOT/cg/h/cghwreg.h**). Most of them implement "set arithmetic": **HW_Asgn**, **HW_CAsgn**, **HW_CEqual**, **HW_COnlyOn**, **HW_COvlap**, **HW_CSubset**, **HW_CTurnOff**, **HW_CTurnOn**, **HW_Equal**, **HW_OnlyOn**, **HW_Ovlap**, **HW_Subset**, **HW_TurnOff**, **HW_TurnOn**.

Sample: Excluding the **EBX** register (required for PIC)

```
hw_reg_set all;
// ...
HW_CTurnOff( all, HW_EBX );
```

There are two levels of code generation for x86: middle level and low level (assuming high level is machine-independent).

High level uses **instruction** and related functions (only some are shown):

```
// Creating new instruction
instruction *MakeUnary( opcode_defs, name *, name *, type_class_def );
instruction *MakeBinary( opcode_defs, name *, name *, name *, type_class_def );
instruction *MakeMove( name *, name *, type_class_def );
// Miscellaneous allocations
name *AllocRegName( hw_reg_set );
name *AllocTemp( type_class_def );
name *AllocIntConst( int );
name *AllocUIntConst( uint );
// Placing the instruction
void AddIns( instruction * );
void PrefixIns( instruction *, instruction * );
void SuffixIns( instruction *, instruction * );
void ReplIns( instruction *, instruction * );
```

Sample: Generating add ebx, 0BABEh in the current block

```
name        *ebx;
instruction *ins;


ebx = AllocRegName( HW_EBX );
ins = MakeBinary( OP_ADD, ebx, AllocIntConst( 0xBABE ), ebx, WD );
AddIns(ins);
```

At the low level, we generate the actual x86 opcodes (once and for all). Transformation from middle level to low level is performed mainly by **i86enc.c**, **i86enc2.c** (**$OWROOT/bld/cg/intel**), **i86enc32.c** (**$OWROOT/bld/cg/intel/386**).

There are several macros for emitting binary opcodes:

```
_Code;
 . . .
_Next;
 . . .
_Emit;
```

Opcodes are inserted using the special functions, e.g.:

```
void LayOpbyte( opcode op );
void LayOpword( opcode op );
void LayReg( hw_reg_set r );
void LayRegOp( name *r );
void LayRMRegOp( name *r );
```

Sample: Generating **PUSHF**

```
_Code;
LayOpbyte( 0x9C );
_Emit;
```

However, there are more digestible functions for common cases, e.g. **GenRegMove()**.

The information above should give the basic knowledge to the developer unfamiliar with CG386. The last uncovered topic is how object files are produced.

Unfortunately the only object format supported is OMF. Therefore many things in CG386 are rigidly bound to OMF structure. OMF output is implemented mostly in **$OWROOT/bld/cg/intel/c/i86obj.c**, **i86esc.c**, and **$OWROOT/bld/cg/c/posixio.c**.

But there is last but one stage, before data became written to the object file. This stage is optimizing (although some optimizations were performed during previous stages). The optimizer (**$OWROOT/bld/cg/c/opt*.c**) has the operations queue. The “trace” below shows intercommunications between the optimizer and OMF output routines (for well-known “Hello, world!” program):

```
#include <stdio.h>


int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Trace:

```
i86obj.c: InitSegDefs()
i86obj.c: DefSegment(id=00000001(1),attr=00000007(7),str="_TEXT",align=00000001(1),use_16=FALSE)
i86obj.c: DefSegment(id=00000002(2),attr=0000001C(28),str="CONST",align=00000004(4),use_16=FALSE)
i86obj.c:
DefSegment(id=00000003(3),attr=0000000C(12),str="CONST2",align=00000004(4),use_16=FALSE)
i86obj.c: DefSegment(id=00000004(4),attr=00000006(6),str="_DATA",align=00000004(4),use_16=FALSE)
i86obj.c: DefSegment(id=0000000B(11),attr=00000002(2),str="_BSS",align=00000004(4),use_16=FALSE)
i86obj.c: ObjInit()
i86obj.c: InitFPPatches()
i86obj.c: FillArray(res,size=00000001(1),starting=00000032(50),increment=00000032(50))
i86obj.c: OutName(name="hello.c",dst)
i86obj.c: NeedMore(arr,more=00000021(33))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutString(name="OS220",dest)
i86obj.c: NeedMore(arr,more=00000005(5))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutModel(dest)
i86obj.c: GetMemModel()
i86obj.c: OutString(name="3fOpd",dest)
i86obj.c: NeedMore(arr,more=00000005(5))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="hello.c",dst)
i86obj.c: NeedMore(arr,more=00000021(33))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/stdio.h",dst)
i86obj.c: NeedMore(arr,more=00000022(34))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/sys/cdefs.h",dst)
i86obj.c: NeedMore(arr,more=00000026(38))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/sys/types.h",dst)
```

```
i86obj.c: NeedMore(arr,more=00000026(38))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/inttypes.h",dst)
i86obj.c: NeedMore(arr,more=00000025(37))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/endian.h",dst)
i86obj.c: NeedMore(arr,more=00000023(35))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/stddef.h",dst)
i86obj.c: NeedMore(arr,more=00000023(35))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/sys/stat.h",dst)
i86obj.c: NeedMore(arr,more=00000025(37))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/stdarg.h",dst)
i86obj.c: NeedMore(arr,more=00000023(35))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutName(name="/usr/lib/dietlibc/include/stdarg-cruft.h",dst)
i86obj.c: NeedMore(arr,more=00000029(41))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FillArray(res,size=00000001(1),starting=00000005(5),increment=00000005(5))
i86obj.c: FillArray(res,size=00000001(1),starting=00000005(5),increment=00000005(5))
i86obj.c: DoSegGrpNames(dgroup_def,tgroup_def)
i86obj.c: GetNameIdx(name="",suff,alloc=)
i86obj.c: GetNameIdx(name="CODE",suff,alloc=)
i86obj.c: GetNameIdx(name="DATA",suff,alloc=)
i86obj.c: GetNameIdx(name="BSS",suff,alloc=)
i86obj.c: GetNameIdx(name="TLS",suff,alloc=)
i86obj.c: GetNameIdx(name="FLAT",suff,alloc=)
i86obj.c: GetNameIdx(name="DGROUP",suff,alloc=)
i86obj.c: OutIdx(value=00000007(7),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FillArray(res,size=00000040(64),starting=00000005(5),increment=00000005(5))
i86obj.c: DoSegment(seg,dgroup_def,tgroup_def,use_16=FALSE)
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SegmentAttr(align=00000001(1),tipe=00000007(7),use_16=FALSE)
i86obj.c: GetNameIdx(name="_TEXT",suff,alloc=)
i86obj.c: SegmentClass(rec)
i86obj.c: DoASegDef(rec,use_16=FALSE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000100(256))
i86obj.c: FillArray(res,size=00000001(1),starting=00000014(20),increment=00000032(50))
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000032(50))
i86obj.c: OutByte(value=00000029(41))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
```

```
i86obj.c: OutIdx(value=00000008(8),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: PickOMF(cmd=00000098(152))
i86obj.c: OutInt(value=0000FE80(65152))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutByte(value=0000004F(79))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: DoSegment(seg,dgroup_def,tgroup_def,use_16=FALSE)
i86obj.c: AskSegIndex(seg=00000002(2))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SegmentAttr(align=00000004(4),tipe=0000001C(28),use_16=FALSE)
i86obj.c: OutGroup(sidx=00000002(2),group_def,index_p)
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: GetNameIdx(name="",suff,alloc=CONST)
i86obj.c: SegmentClass(rec)
i86obj.c: DoASegDef(rec,use_16=FALSE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000100(256))
i86obj.c: FillArray(res,size=00000001(1),starting=00000014(20),increment=00000032(50))
i86obj.c: OutByte(value=000000A9(169))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutIdx(value=00000009(9),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000003(3),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: PickOMF(cmd=00000098(152))
i86obj.c: DoSegment(seg,dgroup_def,tgroup_def,use_16=FALSE)
i86obj.c: AskSegIndex(seg=00000003(3))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SegmentAttr(align=00000004(4),tipe=0000000C(12),use_16=FALSE)
i86obj.c: OutGroup(sidx=00000003(3),group_def,index_p)
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutIdx(value=00000003(3),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: GetNameIdx(name="",suff,alloc=CONST2)
i86obj.c: SegmentClass(rec)
i86obj.c: DoASegDef(rec,use_16=FALSE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000100(256))
i86obj.c: FillArray(res,size=00000001(1),starting=00000014(20),increment=00000032(50))
i86obj.c: OutByte(value=000000A9(169))
```

```
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutIdx(value=0000000A(10),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000003(3),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: PickOMF(cmd=00000098(152))
i86obj.c: DoSegment(seg,dgroup_def,tgroup_def,use_16=FALSE)
i86obj.c: AskSegIndex(seg=00000004(4))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SegmentAttr(align=00000004(4),tipe=00000006(6),use_16=FALSE)
i86obj.c: OutGroup(sidx=00000004(4),group_def,index_p)
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutIdx(value=00000004(4),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: GetNameIdx(name="",suff,alloc=_DATA)
i86obj.c: SegmentClass(rec)
i86obj.c: DoASegDef(rec,use_16=FALSE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000100(256))
i86obj.c: FillArray(res,size=00000001(1),starting=00000014(20),increment=00000032(50))
i86obj.c: OutByte(value=000000A9(169))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutIdx(value=0000000B(11),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000003(3),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: PickOMF(cmd=00000098(152))
i86obj.c: DoSegment(seg,dgroup_def,tgroup_def,use_16=FALSE)
i86obj.c: AskSegIndex(seg=0000000B(11))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SegmentAttr(align=00000004(4),tipe=00000002(2),use_16=FALSE)
i86obj.c: OutGroup(sidx=00000005(5),group_def,index_p)
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutIdx(value=00000005(5),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: GetNameIdx(name="",suff,alloc=_BSS)
i86obj.c: SegmentClass(rec)
i86obj.c: DoASegDef(rec,use_16=FALSE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000100(256))
i86obj.c: FillArray(res,size=00000001(1),starting=00000014(20),increment=00000032(50))
i86obj.c: OutByte(value=000000A9(169))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
```

```
i86obj.c: OutIdx(value=0000000C(12),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000004(4),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: PickOMF(cmd=00000098(152))
i86obj.c: FlushNames()
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: OutIdx(value=00000006(6),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: FlushNames()
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: TellObjNewProc(proc=00000085(133))
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: AskNameCode(hdl=00000049(73),class=00000000(0))
i86obj.c: AskBackSeg()
i86obj.c: SetOP(seg=00000002(2))
i86obj.c: AskSegIndex(seg=00000002(2))
i86obj.c: TellObjNewLabel(lbl=00000000(0))
i86obj.c: SetUpObj(is_data=TRUE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: OutLabel(lbl=0811B9C8(135379400))
i86obj.c: InitPatch()
i86obj.c: FillArray(res,size=0000000C(12),starting=0000000A(10),increment=0000000A(10))
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: AskOP()
i86obj.c: SetUpObj(is_data=TRUE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: OutDBytes(len=0000000F(15),src)
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: NeedMore(arr,more=0000000F(15))
i86obj.c: IncLocation(by=0000000F(15))
i86obj.c: SetBigLocation(loc=0000000F(15))
i86obj.c: SetMaxWritten()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
```

```
i86obj.c: AskNameCode(hdl=0811BE38(135380536),class=00000002(2))
i86obj.c: AskSegID(hdl=0811BE38(135380536),class=00000002(2))
i86obj.c: AskSegPrivate(id=00000002(2))
i86obj.c: AskSegIndex(seg=00000002(2))
i86obj.c: AskSegID(hdl=0811BE38(135380536),class=00000002(2))
i86obj.c: AskSegNear(id=00000002(2))
i86obj.c: AskSegIndex(seg=00000002(2))
i86obj.c: AskBackSeg()
i86obj.c: AskCodeSeg()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: AskCodeSeg()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
i86esc.c: DoSymRef(opnd,val=00000000(0),base=FALSE)
i86esc.c: DoFESymRef(sym=0811BE38(135380536),class=00000002(2),val=00000000(0),fixup=00000001(1))
i86obj.c: AskSegID(hdl=0811BE38(135380536),class=00000002(2))
i86esc.c:
DoRelocRef(sym=0811BE38(135380536),class=00000002(2),seg=00000002(2),val=00000000(0),kind=0000000
0(0))
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
optmain.c: InputOC(oc)
optmain.c: LDone(oc)
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: AskCodeSeg()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86esc.c: CodeHasAbsPatch(code)
i86esc.c: CodeHasAbsPatch(code)
i86esc.c: CodeHasAbsPatch(code)
i86esc.c: OutputOC(oc,next_lbl)
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: AskLocation()
i86esc.c: DoAlignment(len=00000000(0))
i86obj.c: SavePendingLine(new=00000000(0))
i86esc.c: SendBytes(ptr,len=00000000(0))
i86obj.c: SavePendingLine(new=00000000(0))
i86obj.c: OutLabel(lbl=0811BE00(135380480))
```

```
i86obj.c: UseImportForm(attr=00000007(7))
i86obj.c: OutExport(sym=00000085(133))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutObjectName(sym=00000085(133),dest)
i86obj.c: OutName(name="main",dst)
i86obj.c: NeedMore(arr,more=00000005(5))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: OutIdx(value=00000000(0),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: InitPatch()
i86obj.c: FillArray(res,size=0000000C(12),starting=0000000A(10),increment=0000000A(10))
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86esc.c: DumpSavedDebug()
i86esc.c: OutputOC(oc,next_lbl)
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: AskLocation()
i86esc.c: DoAlignment(len=00000000(0))
i86obj.c: SavePendingLine(new=00000000(0))
i86esc.c: SendBytes(ptr,len=00000000(0))
i86obj.c: SavePendingLine(new=00000000(0))
i86obj.c: OutLabel(lbl=0811BC0C(135379980))
i86obj.c: InitPatch()
i86obj.c: FillArray(res,size=0000000C(12),starting=0000000A(10),increment=0000000A(10))
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86esc.c: DumpSavedDebug()
i86esc.c: OutputOC(oc,next_lbl)
i86esc.c: DumpSavedDebug()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86esc.c: ExpandObj(cur,explen=00000009(9))
i86obj.c: OutDBytes(len=00000001(1),src)
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutOffset(value=00000000(0))
i86obj.c: NeedMore(arr,more=00000004(4))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: IncLocation(by=00000001(1))
i86obj.c: SetBigLocation(loc=00000001(1))
i86obj.c: SetMaxWritten()
i86obj.c: OutReloc(seg=00000002(2),class=00000001(1),rel=FALSE)
i86obj.c: AskSegIndex(seg=00000002(2))
i86obj.c: CheckLEDataSize(max_size=0000000C(12),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: DoFix(idx=00000001(1),rel=FALSE,base=00000001(1),class=00000001(1),sidx=00000002(2))
```

```
i86obj.c: NeedMore(arr,more=00000003(3))
i86obj.c: AskIndexRec(sidx=00000002(2))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutDataLong(value=00000000(0))
i86obj.c: OutDataInt(value=00000000(0))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000002(2),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: IncLocation(by=00000002(2))
i86obj.c: SetBigLocation(loc=00000003(3))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: SetMaxWritten()
i86obj.c: OutDataInt(value=00000000(0))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000002(2),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: IncLocation(by=00000002(2))
i86obj.c: SetBigLocation(loc=00000005(5))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: SetMaxWritten()
i86esc.c: OutputOC(oc,next_lbl)
i86esc.c: DumpSavedDebug()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86esc.c: ExpandCJ(oc)
i86obj.c: OutDataByte(value=000000E8(232))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: IncLocation(by=00000001(1))
i86obj.c: SetBigLocation(loc=00000006(6))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SetMaxWritten()
i86esc.c: OutCodeDisp(lbl=0811AF38(135376696),f=00000001(1),rel=TRUE,class=00000008(8))
i86obj.c: UseImportForm(attr=0000000F(15))
i86obj.c: OutImport(sym=00000049(73),class=00000001(1),rel=TRUE)
i86obj.c: FillArray(res,size=00000001(1),starting=00000100(256),increment=00000032(50))
i86obj.c: CheckImportSwitch(next_is_static=FALSE)
i86obj.c: OutName(name="printf",dst)
i86obj.c: NeedMore(arr,more=00000007(7))
i86obj.c: OutIdx(value=00000000(0),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: DumpImportResolve(sym=00000049(73),idx=00000001(1))
i86obj.c: OutSpecialCommon(imp_idx=00000001(1),class=00000001(1),rel=TRUE)
i86obj.c: CheckLEDataSize(max_size=0000000C(12),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: DoFix(idx=00000001(1),rel=TRUE,base=00000002(2),class=00000001(1),sidx=00000000(0))
i86obj.c: NeedMore(arr,more=00000003(3))
i86obj.c: OutIdx(value=00000002(2),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
```

```
i86obj.c: OutIdx(value=00000001(1),dest)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: OutDataLong(value=00000000(0))
i86obj.c: OutDataInt(value=00000000(0))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000002(2),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: IncLocation(by=00000002(2))
i86obj.c: SetBigLocation(loc=00000008(8))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: SetMaxWritten()
i86obj.c: OutDataInt(value=00000000(0))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000002(2),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: IncLocation(by=00000002(2))
i86obj.c: SetBigLocation(loc=0000000A(10))
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: SetMaxWritten()
i86esc.c: OutputOC(oc,next_lbl)
i86esc.c: DumpSavedDebug()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86esc.c: ExpandObj(cur,explen=00000003(3))
i86obj.c: OutDBytes(len=00000003(3),src)
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: NeedMore(arr,more=00000003(3))
i86obj.c: IncLocation(by=00000003(3))
i86obj.c: SetBigLocation(loc=0000000D(13))
i86obj.c: SetMaxWritten()
i86esc.c: OutputOC(oc,next_lbl)
i86esc.c: DumpSavedDebug()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86esc.c: ExpandObj(cur,explen=00000002(2))
i86obj.c: OutDBytes(len=00000002(2),src)
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
i86obj.c: NeedMore(arr,more=00000002(2))
i86obj.c: IncLocation(by=00000002(2))
i86obj.c: SetBigLocation(loc=0000000F(15))
i86obj.c: SetMaxWritten()
i86esc.c: OutputOC(oc,next_lbl)
i86esc.c: DumpSavedDebug()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: OutDataByte(value=000000C3(195))
i86obj.c: SetPendingLine()
i86obj.c: CheckLEDataSize(max_size=00000001(1),need_init=TRUE)
i86obj.c: OutLEDataStart(iterated=FALSE)
```

```
i86obj.c: IncLocation(by=00000001(1))
i86obj.c: SetBigLocation(loc=00000010(16))
i86obj.c: NeedMore(arr,more=00000001(1))
i86obj.c: SetMaxWritten()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: AskCodeSeg()
i86obj.c: SetOP(seg=00000001(1))
i86obj.c: AskSegIndex(seg=00000001(1))
i86obj.c: ObjFini()
i86obj.c: FiniTarg()
i86obj.c: FlushObject()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: GenComdef()
i86obj.c: EjectLEData()
i86obj.c: EjectImports()
i86obj.c: SetPatches()
i86obj.c: SetAbsPatches()
i86obj.c: PickOMF(cmd=000000A0(160))
i86obj.c: PickOMF(cmd=0000009C(156))
i86obj.c: EjectExports()
i86obj.c: PickOMF(cmd=00000090(144))
i86obj.c: FreeObjCache()
i86obj.c: FlushNames()
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: AskIndexRec(sidx=00000001(1))
i86obj.c: KillStatic(arr)
i86obj.c: KillStatic(arr)
i86obj.c: FiniTarg()
i86obj.c: FlushObject()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: GenComdef()
i86obj.c: EjectLEData()
i86obj.c: EjectImports()
i86obj.c: SetPatches()
i86obj.c: SetAbsPatches()
i86obj.c: PickOMF(cmd=000000A0(160))
i86obj.c: EjectExports()
i86obj.c: FreeObjCache()
i86obj.c: AskIndexRec(sidx=00000002(2))
i86obj.c: KillStatic(arr)
i86obj.c: KillStatic(arr)
i86obj.c: FiniTarg()
i86obj.c: FlushObject()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: GenComdef()
i86obj.c: EjectLEData()
i86obj.c: EjectImports()
i86obj.c: EjectExports()
```

```
i86obj.c: FreeObjCache()
i86obj.c: AskIndexRec(sidx=00000003(3))
i86obj.c: KillStatic(arr)
i86obj.c: KillStatic(arr)
i86obj.c: FiniTarg()
i86obj.c: FlushObject()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: GenComdef()
i86obj.c: EjectLEData()
i86obj.c: EjectImports()
i86obj.c: EjectExports()
i86obj.c: FreeObjCache()
i86obj.c: AskIndexRec(sidx=00000004(4))
i86obj.c: KillStatic(arr)
i86obj.c: KillStatic(arr)
i86obj.c: FiniTarg()
i86obj.c: FlushObject()
i86obj.c: SetUpObj(is_data=FALSE)
i86obj.c: CheckLEDataSize(max_size=00000010(16),need_init=FALSE)
i86obj.c: GenComdef()
i86obj.c: EjectLEData()
i86obj.c: EjectImports()
i86obj.c: EjectExports()
i86obj.c: FreeObjCache()
i86obj.c: AskIndexRec(sidx=00000005(5))
i86obj.c: KillStatic(arr)
i86obj.c: KillStatic(arr)
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: KillArray(arr)
i86obj.c: KillStatic(arr)
i86obj.c: FiniAbsPatches()
i86obj.c: EndModule()
```

This trace shows how OMF object file is written in **i86obj.c**. We will refer to this trace in the latter sections.

OMF file is composed of object records. These records contain miscellaneous linking information, e.g.:

EXTDEF	External Names Definition Record (imported symbols)
PUBDEF	Public Names Definition Record (exported symbols)
SEGDEF	Segment Definition Record (describes a logical segment)
GRPDEF	Group Definition Record (segments to be collected together)
FIXUPP	Fixup Record (relocations)
BAKPAT	Backpatch Record (relocations)
LEDATA	Logical Enumerated Data Record (binary code or data)

Actual writing is performed by **void PutObjRec(byte class, byte *buff, uint len)** located in **posixio.c**. For example, class of **LEDATA** is either **0xA0** or **0xA1**.

2.5 OWL

2.5.1 Definition

Abbreviation of "Object Writing Library". Located in **$OWROOT/bld/owl**.

OWL is designed for writing object files in ELF and COFF formats. We are interested mainly in the ELF (**owelf.c**). However, OWL is not an abstract wrapper (like ORL). But rather a set of data structures and functions, useful for creating object files.

2.5.2 Description

OWL is currently used by RISC code generators (Alpha AXP and PowerPC). As mentioned above, OWL is not currently used by CG386.

OWL provides set of useful functions for creating ELF object files. These functions cover sections, symbols, and relocations. For understanding OWL, one can examine **$OWROOT/bld/cg/risc/c/rscobj.c**.

For example, **void OWLEmitReloc(owl_section_handle section, owl_offset offset, owl_symbol_handle sym, owl_reloc_type type)** is intended to add new relocation to the specified section. Relocation type is defined in OWL terms:

```
$OWROOT/bld/owl/h/owl.h

typedef enum {
    OWL_RELOC_ABSOLUTE,       // ref to a 32-bit absolute address
    OWL_RELOC_WORD,           // a direct ref to a 32-bit address
    OWL_RELOC_HALF_HI,        // ref to high half of 32-bit address
    OWL_RELOC_PAIR,           // pair - used to indicate prev hi and next lo
linked
    OWL_RELOC_HALF_LO,        // ref to low half of 32-bit address
    OWL_RELOC_BRANCH_REL,     // relative branch (Alpha: 21-bit; PPC: 14-bit)
    OWL_RELOC_BRANCH_ABS,     // absolute branch (Alpha: not used; PPC: 14-bit)
    OWL_RELOC_JUMP_REL,       // relative jump (Alpha: 14-bit hint; PPC: 24-
bit)
    OWL_RELOC_JUMP_ABS,       // absolute jump (Alpha: not used; PPC:24-bit)
    OWL_RELOC_SECTION_OFFSET, // offset of item within it's section
    // meta reloc
    OWL_RELOC_SECTION_INDEX,  // index of section within COFF file
    OWL_RELOC_TOC_OFFSET,     // 16-bit offset within TOC (PPC)
    OWL_RELOC_GLUE,           // location of NOP for GLUE code
    OWL_RELOC_FP_OFFSET,      // cheesy hack for inline assembler
} owl_reloc_type;
```

These abstract types are mapped to ELF relocation types:

```
$OWROOT/bld/owl/c/owreloc.c

static Elf32_Word elfRelocTypes386[] = {
    R_386_NONE,
    R_386_32,
    R_386_NONE,
    R_386_NONE,
    R_386_NONE,
    R_386_PC32,
    R_386_NONE,
    R_386_NONE,
    R_386_NONE,
    R_386_32,
    R_386_32,
    R_386_GOT32,
    R_386_NONE,
};
```

As shown above, OWL is intended primarily to RISC support, so many 386 ABI features are missing or incomplete.

3. Porting Open Watcom C Compiler and Linker to Linux

The porting task was originally defined as:

- Add PIC support to the compiler.
- Implement building of shared objects (both PIC and PDC).
- Implement using of existing shared objects.

This task was defined with the assumption Open Watcom is already able to build ELF files (i.e. suitable for Linux). This is almost true, but there are two problems:

- Some bugs (in **open_watcom_devel_1.1.7**) causing problems in building ELF executables (even "Hello, world!").
- The only object file format produced by CG386 is OMF.

The first problem is, of course, temporary. After fixing the mentioned bugs, it is possible to build ELF executable from OMF and ELF object files and e.g. **dietlibc** library.

However, the second problem is more serious. It affects the perspective of PIC implementation (and therefore, building of "real" shared objects). The corresponding issues are described in the latter sections.

3.1 Position-Independent Code

PIC stands for Position-Independent Code. The functions in a shared library may be loaded at different addresses in different programs, so the code in the shared object must not depend on the address (or position) at which it is loaded. Fortunately, on x86 platform all jumps are PC-relative (except for the indirect ones). There are, however, some problems with:

- functions exported by a shared object;
- indirect function calls, i.e. **(*f)()**;
- global variables (including **static** ones).

These problems are solved (in 386 ABI) mostly by introducing special relocation types. These relocation types are specific to ELF object files, there are no their equivalents in OMF.

There are three possible workarounds:

- introducing OMF extensions for PIC support;
- adding ELF output to CG386 (using OWL);
- writing new code generator with ELF output (based on CG386), like RISC ones.

The first approach is the simplest from implementation perspective. But we will get a non-standard object file format, alien to both Linux and Windows worlds. Therefore such approach should be ommited.

The third approach seems too hard to implement, since CG386 is the most complex code generator. And it seems impractical to have two branches of CG386 that differ only in the output format.

So the second approach is the best option. There are three subtasks needed for PIC support:

- introducing new command line switches in **wcc386** (for ELF and PIC);
- implementing output of ELF object files in CG386;
- implementing PIC (according to 386 ABI) in CG386.

Of course, changes to Open Watcom Linker are needed as well. But **wlink** is described in other sections. Moreover, we can use **ld** to build shared object from ELFs produced by **wcc386**.

3.1.1 Command Line Switches

There are no ELF and PIC switches in Open Watcom C Compiler. In **gcc**, ELF is default format of object files, and PIC generation is turned on by either **-fPIC** or **-fpic**.

Since command line of **wcc386** differs from **gcc** very much, we may follow the "Watcom style". For ELF, perhaps the best option is **-elf**. The "el" prefix is free, since the "nearest" options are **-ei** and **-em**. And this choice is logical, because the option **-ez** stands for "generate PharLap EZ-OMF object files".

For PIC, the GNU style seems unacceptable, since "fp" prefix is intended for floating-point options. Especially, **-fpi** means "inline 80x87 instructions with emulation". Like in the ELF case, simply **-pic** may be acceptable (however, "p" prefixes preprocessor options). As alternative, **-zpic** seems a good choice, since "z" groups very specific options. There is also **-re** switch (already implemented). This switch is mapped to **POSITION_INDEPENDANT** option in CG386, but nothing reasonable is performed when it is turned on.

Finally, our switch should be passed from **wcc386** to CG386. The interesting files are: **$OWROOT/bld/cc/c/coptions.c, cgen2.c, $OWROOT/bld/cg/h/cgswitch.h, and $OWROOT/bld/cg/intel/h/cgi86swi.h**. In the compiler, switches are stored in **CompFlags** variable. Other important variables are **GenSwitches** and (especially) **TargetSwitches**.

Switches are passed to CG386 in **cgen2.c**:

```
void DoCompile()

{

            // ...

            cgi_info = BEInit( GenSwitches, TargetSwitches, OptSize,
ProcRevision );

            // ...

}
```

3.1.2 ELF Object Files

Since Open Watcom already contains OWL with ELF support, it is planned to use this library in CG386. Both CG386 and OWL were described briefly in the previous sections.

Many things in CG386 are rigidly bound to OMF structure. OMF output is implemented mostly in **$OWROOT/bld/cg/intel/c/i86obj.c**, **i86esc.c**, and **$OWROOT/bld/cg/c/posixio.c**. However, any object file format defines virtually the same objects: groups, segments, symbols, relocations, etc. The biggest conceptual difference between OMF and ELF is relocation handling. But the opposite problem was successfully solved in ORL and WLCore. So we can implement the same "mapping" approach in CG386, avoiding harmful changes to the complicated code generator.

The sample trace (see section CG386) shows how OMF object file is created.

Code and data (i.e. binary payload) are written by **EjectLEData()**. Although there are many calls of this function in the trace, data are written when the following condition is true: **obj->data.used > CurrSeg->data_prefix_size**. Instead of calling **PutObjRec()**, we will call **OWLEmitData()**. Note that fix-ups are written in **EjectLEData()** as well. So **OWLEmitReloc()** should be used to write relocations.

Sample mapping between **i86obj.c** and OWL is shown below. Each entry means that we *can* use specified OWL function for OMF task, so there is no direct correspondence between columns.

OMF	OWL
DefSegment()	OWLSectionInit()
EjectImports()	OWLEmitImport()
EjectExports()	OWLEmitExport()
OutLabel()	OWLSymbolInit()

RISC object code, located in **$OWROOT/bld/$OWROOT/bld/rscobj.c**. can be used as a reference.

Unfortunately OWL is RISC-oriented, so missing features should be added. There are some relocation types missing in the current OWL. These relocations are described in the next section. Abstract relocation types are defined in **owl.h**. The mapping between OWL relocations and 386 ABI is defined in **owreloc.c**:

```
static Elf32_Word elfRelocTypes386[] = {
    R_386_NONE,
    R_386_32,
    R_386_NONE,
    R_386_NONE,
    R_386_NONE,
    R_386_PC32,
    R_386_NONE,
    R_386_NONE,
    R_386_NONE,
    R_386_32,
    R_386_32,
    R_386_GOT32,
    R_386_NONE,
};
```

Since some relocations are 386-specific, the corresponding constants to both files should be added. In addition, new fixup flags are needed for the mapping between OMF-style fixups and OWL relocations. Extending fixups seems to be the hardest part of this subtask.

The only remark is that OWL in **open_watcom_devel_1.1.7** seemed to be under development, i.e. some features are incomplete.

3.1.3 PIC Generation

GOT base register

The **EBX** register serves as the global offset table base register for position-independent code. So this register should be excluded from normal code generation.

Register macros were described in CG386 section. The following template illustrates turning off **EBX**.

```
$OWROOT/bld/cg/intel/386/c/386rgtbl.c
extern  hw_reg_set      FixedRegs() {
/***********************************
    return the set of register which may not be modified within this routine
*/

    hw_reg_set  fixed;
    // ...
    HW_CTurnOn( fixed, HW_EBX ); // PIC
    return( fixed );
}

extern  hw_reg_set      AllCacheRegs() {
/***************************************
    return the set of all registers that could be used to cache values
*/

    hw_reg_set all;
    // ...
    HW_CTurnOff( all, HW_EBX ); // PIC
    return( all );
}
```

Position-Independent Function Prologue

```
prologue:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $80, %esp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        call    .L1
.L1:    popl    %ebx
        addl    $_GLOBAL_OFFSET_TABLE_+[.-.L1], %ebx
```

The **call** instruction pushes the absolute address of the next instruction onto the stack.

Consequently, the **popl** instruction pops the absolute address of **.L1** into register **%ebx**.

The last instruction computes the desired absolute value into **%ebx**. This works because **_GLOBAL_OFFSET_TABLE_** in the expression gives the distance from the **addl** instruction to the global offset table; **[.-.L1]** gives the distance from **.L1** to the **addl** instruction. Adding their sum to the absolute address of **.L1**, already in **%ebx**, gives the absolute address of the global offset table.

The last line seems a bit complicated, since there is address calculation. But actually this line should be **add $0x3,%ebx**, where immediate is the explicit addend for **R_386_GOTPC** relocation. Note that code generator should create the undefined symbol **_GLOBAL_OFFSET_TABLE_**, if **R_386_GOTPC** encountered.

Prologues are handled in **$OWROOT/bld/cg/intel/c/i86proc.c**, so needed code should be added to **void GenProlog(void)**.

Template:

```
    pointer lbl;
    // ...
            AllocStack();
            AdjustPushLocals();
            // PIC
            lbl = AskForNewLabel();
            GenCallLabel( lbl );
            CodeLabel( lbl, 0 );
            QuickSave( HW_EBX, OP_POP );
            GenRegAdd( HW_EBX, 3 );
            // Add relocation R_386_GOTPC
            GenKillLabel( lbl );
            ///PIC
```

In addition, register **EBX** should be saved in the stack. There is variable **to_push** in **GenProlog()**, so the needed code is: **HW_CTurnOn(to_push, HW_EBX)**.

Moreover, 386 ABI notes that **EBX**, **ESI**, and **EDI** should be saved in the stack, for both PIC and PDC.

Of course, PIC actions depend on CG386 switch for PIC. So conditional processing as well should be also added.

Position-Independent Function Epilogue

All registers previously saved in stack (see above) should be resotred.

Although epilogue is created in **void GenEpilog(void)**, the interesting function is **void DoEpilog(void)**. Both are defined in **$OWROOT/bld/cg/intel/c/i86proc.c**.

There is variable **to_pop**, defining the register set to be popped.

PIC Function Calls

Function calls are handled in **$OWROOT/bld/cg/intel/c/i86call.c**. There is also important function **void AddCallIns(instruction *ins, cn call)** located in **$OWROOT/bld/cg/c/bldcall.c**.

Since ELF-specific relocations are not implemented yet, there is no code template. However, the task is simple. For PDC, the target address has **R_386_PC32** relocation. For PIC, this relocation should be **R_386_PLT32**. The corresponding GNU assembler line is **call function@PLT**.

The information above covers direct function calls. Indirect function calls are kind of PIC data access described below.

PIC Data Access

This task covers accessing the global data (including **extern** and **static**). Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' offsets into the global offset table. Combining the offset with the global offset table address in **EBX** gives the absolute address of the table entry holding the desired address.

Sample	PDC	PIC

`extern int src;`
`extern int dst;`
`extern int *ptr;`
`ptr = &dst;`
	`.globl src, dst, ptr`
	
	`movl $dst, ptr`
		`.globl src, dst, ptr`
		
		`movl ptr@GOT(%ebx), %eax`
		`movl dst@GOT(%ebx), %edx`
		`movl %edx, (%eax)`

`*ptr = src;`	`movl ptr, %eax`
	`movl src, %edx`
	`movl %edx, (%eax)`
		`movl ptr@GOT(%ebx), %eax`
		`movl (%eax), %eax`
		`movl src@GOT(%ebx), %edx`
		`movl (%edx), %edx`
		`movl %edx, (%eax)`

Although references like **name@GOT** seem complicated, their meaning is simple, e.g. **mov 0x0(%ebx), %eax**, where **0x0** is addend for relocation **R_386_GOT32**, associated with symbol **ptr**. For PDC, relocation type is **R_386_32**, and generated code is much simpler.

Finally, position-independent references to static data may be optimized. Because **EBX** holds a known address, the global offset table, a program may use it as a base register. External references should use the global offset table entry, because dynamic linking may bind the entry to a definition outside the current object file's scope. For **static** variables, the PIC code will be the following:

```
leal    ptr@GOTOFF(%ebx), %eax
leal    dst@GOTOFF(%ebx), %edx
movl    %edx, (%eax)
movl    ptr@GOTOFF(%ebx), %eax
movl    src@GOTOFF(%ebx), %edx
movl    %edx, (%eax)
```

Again, references **name@GOTOFF** actually correspond to relocations **R_386_GOTOFF**, where relocation symbol is the segment (e.g. **.bss**), and implicit addend is offset of **name** in this segment.

There is no code template for data access. This task is most complicated, so additional investigation is needed. PIC data access might affect the common code generator (not only CG386). In general, PIC global variable should be treated as pointer to the actual address instead of address itself.

One potentially useful function is **AddGlobalIndex()** located in **$OWROOT/bld/cg/intel/386/c/386opseg.c**. This function adds **EBX** to every memory reference.

Summary

For PIC support, code generator should be able to produce some specific relocations (in addition to **R_386_32** and **R_386_PC32**). These relocations are summarized below (now from the perspective of link editor).

R_386_GOT32	This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.
R_386_GOTOFF	This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.
R_386_GOTPC	This relocation type resembles **R_386_PC32**, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is **_GLOBAL_OFFSET_TABLE_**, which additionally instructs the link editor to build the global offset table.
R_386_PLT32	This relocation type computes the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

3.1.4 Notes

The information presented in the sections above should not be treated as retelling of 386 ABI. It should be used together with ABI documentation. Some details are omitted. During the porting work, developer should refer to ABI and other documentation; perform analysis using **objdump** and **readelf**; etc.

3.2 Building Shared Objects

This section describes the changes to Open Watcom Linker, needed for building shared libraries (PIC and PDC).

3.2.1 Linker Command Line

Fortunately, the command line option for building a shared object is already implemented in Open Watcom Linker. In such case, one should execute the linker this way: **wlink form ELF DLL ...**

DLL stands for Dynamic Linking Library that is shared object in the Linux world. One can check that ELF shared object was requested by examining **FmtData**:

```
if( (FmtData.type & MK_ELF) && FmtData.dll ) {
    // Do something...
}
```

3.2.2 ELF Header

Currently, LoadELF is able to produce only executable files (**ET_EXEC**). Shared objects have type **ET_DYN**. The following change is needed:

```
$OWROOT/bld/wl/c/loadelf.c
static void SetHeaders( ElfHdr *hdr )
/**********************************/
{
    // ...
    hdr->eh.e_type = FmtData.dll ? ET_DYN : ET_EXEC;
    // ...
}
```

3.2.3 Segments and Sections

This is only a sample. Coding tasks are described in latter sections.

ELF executables created by **wlink** are organized in the following way (output from **readelf -a**):

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x00000000 0x????? 0x????? R E 0
  LOAD           0x?????? 0x080????? 0x00000000 0x????? 0x????? R E 0x1000
  LOAD           0x?????? 0x080????? 0x00000000 0x????? 0x????? RW  0x1000


 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
   02     .data .bss
```

Shared objects created by **ld** are organized in the following way (complete example):

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x????? 0x????? R E 0x1000
  LOAD           0x?????? 0x???????? 0x???????? 0x????? 0x????? RW  0x1000
  DYNAMIC        0x?????? 0x???????? 0x???????? 0x????? 0x????? RW  0x4


 Section to Segment mapping:
  Segment Sections...
   00     .hash .dynsym .dynstr .rel.dyn .rel.plt .plt .text .rodata
   01     .data .dynamic .got .bss
   02     .dynamic
```

We will refer to these samples from other sections of this document.

3.2.4 Program Headers

Unnecessary **PT_PHDR**

Since shared object is not a program, the program header entry **PT_PHDR** is not needed. Program headers are allocated in **SetHeaders()**, and the first element is always **PT_PHDR**.

Template:

```
static void SetHeaders( ElfHdr *hdr )
/************************************/
{
    hdr->eh.e_phnum = NumGroups + (FmtData.dll ? 0 : 1);
    // ...
    if( !FmtData.dll ) {
        hdr->ph->p_type = PT_PHDR;
        hdr->ph->p_offset = sizeof(Elf32_Ehdr);
        hdr->ph->p_vaddr = sizeof(Elf32_Ehdr) + FmtData.base;
        hdr->ph->p_paddr = 0;
        hdr->ph->p_filesz = hdr->ph_size;
        hdr->ph->p_memsz = hdr->ph_size;
        hdr->ph->p_flags = PF_R | PF_X;
        hdr->ph->p_align = 0;
    }
    // ...
```

```
}
```

But **PT_PHDR** is assumed in **WriteELFGroups()**, so that function should be changed as well: **ph = hdr->ph + (FmtData.dll ? 0 : 1)**.

Necessary **PT_DYNAMIC**

Although **PT_PHDR** is never used in shared objects, **PT_DYNAMIC** is always used there. This program header specifies dynamic linking information.

Therefore we can leave **hdr->eh.e_phnum = NumGroups + 1**, for executable and shared object. Following the GNU convention, we will place dynamic segment after other segments.

3.2.5 Dynamic Section

The **PT_DYNAMIC** segment contains the **.dynamic** section. This section (with type **SHT_DYNAMIC**) contains an array of the following structures.

```
$OWROOT/bld/watcom/h/exeelf.h

typedef struct {
    Elf32_Sword         d_tag;
    union {
        Elf32_Word      d_val;
        Elf32_Addr      d_ptr;
    } d_un;
} Elf32_Dyn;


// dynamic array tags


#define DT_NULL         0
#define DT_NEEDED       1               // name of a needed library
#define DT_PLTRELSZ     2               // size of reloc entries for PLT
#define DT_PLTGOT       3               // address with PLT or GOT
#define DT_HASH         4               // symbol hash table address
#define DT_STRTAB       5               // string table address
#define DT_SYMTAB       6               // symbol table address
#define DT_RELA         7               // address of reloc table with addends
#define DT_RELASZ       8               // size of the DT_RELA table
#define DT_RELAENT      9               // size of a DT_RELA entry
#define DT_STRSZ        10              // size of the string table
#define DT_SYMENT       11              // size of a symbol table entry
#define DT_SONAME       14              // shared object name
```

```
#define DT_REL          17              // address of reloc table without
addends
#define DT_RELSZ        18              // size of the DT_REL table
#define DT_RELENT       19              // size of a DT_REL entry
#define DT_PLTREL       20              // type of reloc entry for PLT
#define DT_DEBUG        21              // for debugging information
#define DT_JMPREL       23              // reloc entries only with PLT
```

This section should reside in the data segment. We can create this section at the end of first linker pass (as segment in **DGROUP**). This section will be written later using LoadELF. Additionally, the program header **PT_DYNAMIC** should be updated.

3.2.6 Dynamic Symbols

Shared objects contain two symbol tables (i.e. sections): normal symbol table (**SHT_SYMTAB**), and dynamic symbol table (**SHT_DYNSYM**). The name of dynamic symbol table is **.dynsym** instead of **.symtab**.

Both are generally the same, but dynamic table does not contain local symbols (except sections). Of course, the corresponding string table should be created (**.dynstr** instead of **.strtab**). For shared objects, section **.hash** is related to dynamic symbol table.

Some changes are needed to **WriteElfSymTable()**, located in **loadelf2.c**. Function **WriteSHStrings()** from **loadelf.c** should be changed as well. These changes include providing virtual addresses of the corresponding sections (there is no memory allocation for normal symbol table), and supporting different section names and types. Like normal symbol table, two variables are needed for dynamic table:

```
$OWROOT/bld/wl/h/loadelf.c
static stringtable      SymStrTab;
static ElfSymTable *    ElfSymTab;
static stringtable      DynStrTab; // new
static ElfSymTable *    DynSymTab; // new
```

The corresponding changes are needed to **void InitSections(ElfHdr *hdr)** (i.e. allocating **.dynsym** and **.dynstr**), and to **void ChkElfData(void)** (i.e. initializing dynamic symbol table). ELF handle should be modified as well:

```
$OWROOT/bld/wl/h/loadelf2.h
typedef struct {
    Elf32_Ehdr eh;
    // ...
    stringtable secstrtab;
    struct {
        int        secstr;  // Index of strings section for section names
```

```
        int         grpbase; // Index base for Groups in section
        int         grpnum;  // Number of groups
        int         relbase; // Index base for relocation sections
        int         relnum;  // number of relocations
        int         symstr;  // Index of symbol's string table
        int         symtab;  // Index of symbol table
        int         symhash; // Index of symbol hash table
        int         dynsym;  // Index of dynamic symbol's string table
        int         dynstr;  // Index of dynamic symbol table
        int         dbgbegin;// Index of first debug section
        int         dbgnum;  // Number of debug sections
    } i;  // Indexes into sh
    unsigned_32 curr_off;
} ElfHdr;
```

Finally, the dynamic array should be updated (i.e. **DT_HASH**, **DT_STRTAB**, **DT_SYMTAB**, **DT_STRSZ**, and **DT_SYMENT**).

3.2.7 Dynamic Relocations

Relocations are written using **WriteRelocsSections()**. However, a shared object should contain single relocation table **.rel.dyn**. So the new function, say **WriteDynRelocsSection()**, is needed. This function should merge all relocations into single table. It should update **.dynamic** section as well (i.e. either **DT_RELA**, **DT_RELASZ**, **DT_RELAENT** or **DT_REL**, **DT_RELSZ**, **DT_RELENT**). Note that the current implementation of **WriteRelocsSections()** generates only “rela” relocations (i.e. explicit addend).

Template

```
// Initialize sh and its fields
// ...
AddSecName( hdr, sh, ".rela.dyn" );
for( group = Groups; group != NULL; group = group->next_group ) {
    relocs = group->g.grp_relocs;
    if( relocs != NULL ) {
        size = RelocSize( relocs );
        sh->sh_size += size;
        DumpRelocList( relocs );
        hdr->curr_off += size;
    }
    currgrp++;
```

```
}
// Update the dynamic array
// ...
```

3.2.8 Global Offset Table

When link editor encounters one of the following relocation types:

- **R_386_GOT32**
- **R_386_GOTOFF**
- **R_386_GOTPC**

it should build the Global Offset Table. Additionally **wlink** should process this relocation types according to 386 ABI.

GOT is defined as **Elf32_Addr _GLOBAL_OFFSET_TABLE_[]**.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol **_DYNAMIC**. Entries one and two in the global offset table also are reserved.

One should add support of these relocation types to both ORL and WLCore. For ORL, introduce new constants in **orlglobl.h**, e.g.: **ORL_RELOC_TYPE_GOT_32**, **ORL_RELOC_TYPE_GOT_OFF**, **ORL_RELOC_TYPE_GOT_REL**. Then extend the mapping between ELF and ORL (**elflwlv.c**).

Then the mapping between **ORL_** and **FIX_** should be added to **DoReloc()**, **objorl.c**. Of course, new **FIX_** constants are needed as well (**obj2supp.h**).

Relocation processing is performed in **obj2supp.c**. Some preprocessing is performed in **objorl.c** as well.

Relocation types mentioned above are processed in the following way.

A This means the addend used to compute the value of the relocatable field.

G This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution.

GOT This means the address of the global offset table.

S This means the value of the symbol whose index resides in the relocation entry.

R_386_GOT32	G + A - P
R_386_GOTOFF	S + A - GOT
R_386_GOTPC	GOT + A - P

As shown in the table, **R_386_GOT32** and **R_386_GOTPC** are processed very close to **R_386_PC32** (**S + A - P**). This mean both are **FIX_OFFSET_32 | FIX_REL**. Similarly, **R_386_GOTOFF** should be **FIX_OFFSET_32**. Of course, additional **FIX_** flags are needed to distinguish them for further processing in **obj2supp.c**.

We can build the GOT during ORL conversion, i.e. in **DoReloc()**. During this phase, symbol offsets into the GOT are calculated.

At the end of first linker pass, we can create the **.got** section (i.e. segment in **DGROUP**). At this time, **_GLOBAL_OFFSET_TABLE_** symbol should be defined as well. This allows creating the GOT with minimal changes to the source code.

For unresolved external symbols, GOT entries are needed as well. Linker should create **R_386_GLOB_DAT** relocations for such GOT entries. These relocations are associated with unresolved symbols.

Another relocation type that can appear in shared object is **R_386_RELATIVE**. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Such relocations are created from **R_386_GOT32**, if the corresponding symbol is not external.

In this section, only GOT aspects related with data were described. Code aspects are described in the next section.

Finally, the dynamic array should be updated (**PLTGOT**).

Note that there is PowerPC TOC implementation in Open Watcom Linker. TOC is close to GOT in some sense, but in general it is different thing. However, developer should take a look at existing TOC implementation, since it contains some useful ideas.

3.2.9 Procedure Linkage Table

PLT is like GOT in some sense, but it is associated with PIC code instead of PIC data. Although 386 ABI defines PLT for PDC and PIC, only PIC PLT is needed for our current task:

```
.PLT0:  pushl   4(%ebx)
        jmp     *8(%ebx)
        nop; nop
        nop; nop

.PLT1:  jmp     *name1@GOT(%ebx)
        pushl   $offset
        jmp     .PLT0@PC

.PLT2:  jmp     *name2@GOT(%ebx)
        pushl   $offset
        jmp     .PLT0@PC

        ...
```

.PLT0@PC in each entry means the distance between the corresponding **jmp** and **.PLT0**, since x86 jumps are PC-relative.

The GOT entry should be created for each PLT entry. Such GOT entry should contain the address of the following **pushl** instruction, not the real address of e.g. **name1**. Thus **name1@GOT** means the offset of the corresponding GOT entry.

A new **R_386_JUMP_SLOT** relocation should be created. Its offset will specify the global offset table entry used in the previous **jmp** instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, e.g. **name1**. Instructions **pushl $offset** pushes the offset of such relocation in the PLT relocation table (**rel.plt**).

When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Therefore these entries are reserved.

Since PLT contains instruction opcodes, an implementation template is presented for advice:

```
typedef struct pltent1 {
    unsigned_16 push_ins;
    unsigned_32 push_ofs;
    unsigned_16 jmp_ins;
    unsigned_32 jmp_ofs;
    unsigned_32 nops;
} PLTENT1;

typedef struct pltentn {
    unsigned_16 jmp1_ins;
    unsigned_32 jmp1_ofs;
    unsigned_8  push_ins;
    unsigned_32 push_ofs;
    unsigned_8  jmp2_ins;
    unsigned_32 jmp2_ofs;
} PLTENTN;

typedef union pltent {
    PLTENT1     first;
    PLTENTN     entry;
} PLTENT;

typedef struct plt {
    unsigned    nentries;
    PLTENT      *entries;
} PLT;

void InitPLT ( PLT *plt ) {
    plt->entries = AllocMem( sizeof( PLTENT ) );
```

```
    plt->entries[0].first.push_ins = 0xB3FF;
    plt->entries[0].first.push_ofs = 0x00000004;
    plt->entries[0].first.jmp_ins  = 0xA3FF;
    plt->entries[0].first.jmp_ofs  = 0x00000008;
    plt->entries[0].first.nops     = 0x90909090;
    plt->nentries = 1;
}

void Add2PLT ( PLT *plt, unsigned_32 gotoff) {
    static unsigned_32 reloff = 0;
    plt->entries = ReallocMem( plt->entries, (plt->nentries + 1) * sizeof(
PLTENT ) );
    plt->entries[plt->nentries].entry.jmp1_ins = 0xA3FF;
    plt->entries[plt->nentries].entry.jmp1_ofs = gotoff;
    plt->entries[plt->nentries].entry.push_ins = 0x68;
    plt->entries[plt->nentries].entry.push_ofs = reloff; reloff += sizeof(
Elf32_Rel );
    // Add R_386_JUMP_SLOT relocation for push_ofs (somehow)...
    plt->entries[plt->nentries].entry.jmp2_ins = 0xE9;
    plt->entries[plt->nentries].entry.jmp2_ofs = -0x10 - plt->nentries *
sizeof( PLTENT );
    plt->nentries++;
}
```

When relocation **R_386_PLT32** is encountered, the linker should create new PLT entry for the corresponding symbol (but only if its type is **STT_FUNC**). For further references to the same symbol, we will refer to the previously created PLT entry. **R_386_PLT32** relocations are processed as **L + A - P**, where **L** means the place (section offset or address) of the procedure linkage table entry for a symbol, **A** and **P** were defined in the previous section. The corresponding ORL type, **ORL_RELOC_TYPE_PLT_32**, is already defined (but not implemented yet). This is relative type, so the mapping should include **FIX_OFFSET_32 | FIX_REL**. Source files and functions participating in relocation process were described in the previous section.

At the end of first linker pass, we can create the **.plt** section (i.e. segment in **AUTO** group). This allows creating the PLT with minimal changes to the source code. Note that the separated relocation section (**.rel.plt**) is needed for PLT relocations. This relocation table should also reside in the code segment.

Finally, the dynamic array should be updated (**PLTRELSZ**, **PLTREL**, and **JMPREL**).

3.2.10 Notes

The information presented in the sections above should not be treated as retelling of 386 ABI. It should be used together with ABI documentation. Some details are omitted, e.g. section flags for the dynamic section. During the porting work, developer should refer to ABI and other documentation; perform analysis using **objdump** and **readelf**; etc.

Note that symbol types (**STT_**) are very important for dynamic linking tasks. For example, **STT_FUNC** is closely related with PLT. The current implementation (i.e. **open_watcom_devel_1.1.7**) sometimes loses symbol types, so such issues need to be fixed.

3.3 Using Shared Objects

This section describes the changes to Open Watcom Linker, needed for using existing shared libraries (PIC and PDC). Note that a shared library may use other shared libraries as well.

Since many things are related to building shared libraries (which is covered in the provious sections of this document), this section is sufficiently short.

3.3.1 Reading Shared Objects

Shared object is another kind of ELF object file. ORL is able to read ELF object files. Some features related to shared objects are implemented as well. Thus **wlink** fails (i.e. **Segmentation fault**) when one tries to link a shared object. ORL should be reviewed and fixed in respective to these issues.

Additionally, the linker should collect the names of shared objects for further processing (see "Needed Libraries" below). If this list is non-empty, the linker should perform some tasks described in the further sections.

3.3.2 Program Interpreter

The additional program header **PT_INTERP** is needed for an executable that uses shared object(s). It specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. For Linux, the program interpreter is **/lib/ld-linux.so.2**

The needed changes in LoadELF are simple and obvious.

3.3.3 Required Libraries

The additional element of the dynamic array (i.e. **.dynamic** section) is needed. When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in **DT_NEEDED** entries of the dynamic structure) tell what shared objects are needed to supply the program's services.

DT_NEEDED holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the **DT_STRTAB** entry. The dynamic array may contain multiple entries with this type. These entries' relative order is significant, though their relation to entries of other types is not.

Dynamic array is described in the previous sections.

3.3.4 Global Offset Table

The GOT processing is described in previous section. If any specific relocation is encountered, the linker should resolve them and create the Global Offset Table. See also the next section.

3.3.5 Procedure Linkage Table

The PLT processing is described in previous section. If any specific relocation is encountered, the linker should resolve them and create the Procedure Linkage Table.

There is, however, one important case not covered in the previous sections. If PDC shared object is needed for the program, the linker creates PDC PLT. Its format differs from PIC PLT:

```
.PLT0:  pushl   got_plus_4
        jmp     *got_plus_8
        nop; nop
        nop; nop

.PLT1:  jmp     *name1_in_GOT
        pushl   $offset
        jmp     .PLT0@PC

.PLT2:  jmp     *name2_in_GOT
        pushl   $offset
        jmp     .PLT0@PC

        ...
```

Here **got_plus_4** and **got_plus_8** specify explicit addresses of the second and third GOT entries, respectively. Similarly, **name1_in_GOT** specifies address of the GOT entry for **name1**.

Instead of implementation template (very similar to PIC one), a sample disassembly is presented:

```
08048224 <.plt>:
 8048224:       ff 35 b4 93 04 08       pushl  0x80493b4    ;&GOT[1]
 804822a:       ff 25 b8 93 04 08       jmp    *0x80493b8   ; GOT[2]
 8048230:       00 00
 8048232:       00 00
 8048234:       ff 25 bc 93 04 08       jmp    *0x80493bc   ; GOT[3]
 804823a:       68 00 00 00 00          push   $0x0
 804823f:       e9 e0 ff ff ff          jmp    8048224
 8048244:       ff 25 c0 93 04 08       jmp    *0x80493c0   ; GOT[4]
 804824a:       68 08 00 00 00          push   $0x8
 804824f:       e9 d0 ff ff ff          jmp    8048224
```

3.3.6 Notes

During the porting work, developer should refer to ABI and other documentation; perform analysis using **objdump** and **readelf**; etc.

Note that symbol types (**STT_**) are very important for dynamic linking tasks. For example, **STT_FUNC** is closely related with PLT. The current implementation (i.e. **open_watcom_devel_1.1.7**) sometimes loses symbol types, so such issues need to be fixed.

4. Existing Problems

Several problems exist in **open_watcom_devel_1.1.7**, more precely, in the linker. So one is unable to make even the “Hello, world!” program.

NOTE: By the time of the final revision of this document all the problems mentioned in this section were fixed in the Open Watcom Perforce depot therefore altering an estimated time requirements. (See estimation section).

4.1.1 Support of R_386_PC32 relocations

After linking, the relocated values are 4 less than they should be.

Gcc, nasm, and other Linux compilers typically generate the following:

```
e8 fc ff ff ff    call  somefunc
```

(**0xfffffffc** is the implicit addend for **R386_PC32** relocation).

Watcom C typically generates the following (of course, in OMF format):

```
e8 00 00 00 00    call  somefunc
```

```
$OWROOT/bld/wl/c/obj2supp.c
static bool CheckSpecials( fix_data *fix, frame_spec *targ )
/*********************************************************/
{
      .     .     .
      if( !(fix->type & FIX_REL) ) return FALSE;
      .     .     .
      fixsize = CalcFixupSize( fix->type );
      off -= fixsize;
      .     .     .
}
```

This algorithm introduces our 4-byte error. Such correction isn't needed for **ELF R386_PC32**, since implicit addend is specified (**0xfffffffc == -4**).

QUICK FIX: Offset correction should be disabled in case implicit addend was specified.

```
$OWROOT/bld/wl/c/obj2supp.c
static bool CheckSpecials( fix_data *fix, frame_spec *targ )
/*********************************************************/
{
      .     .     .
      if( !(fix->type & FIX_REL) ) return FALSE;
      .     .     .
      fixsize = CalcFixupSize( fix->type );
      if( fix->type & FIX_ADDEND_ZERO ) off -= fixsize; // quickfix #01
      .     .     .
}
```

NOTE. This is temporary solution. New ORL relocation type (or option) is needed for a more accurate fix. This bug was already fixed in the development source tree at the moment of writing this SRS.

4.1.2 Support of STT_NOTYPE symbols

Two of symbol types defined in ABI:

STT_NOTYPE The symbol's type is not specified.

STT_FUNC The symbol is associated with a function or other executable code.

Many of "real-life" ELF object files has symbols of **STT_NOTYPE**, e.g. **_start** in dietlibc's **start.o**. When linking that sort of object files, Open Watcom Linker complains such symbols are not found. This error is fatal.

ORL treats **STT_NOTYPE** and unknown symbol types as **ORL_SYM_TYPE_NONE**. This (somehow) confuses the linker.

```
$OWROOT/bld/orl/elf/c/elflwlv.c
            default:
                current->type = ORL_SYM_TYPE_NONE; // ?
```

QUICK FIX: If symbol's associated section (i.e. **st_shndx**) looks like executable, treat that symbol as **ORL_SYM_TYPE_FUNCTION**.

```
$OWROOT/bld/orl/elf/c/elflwlv.c
orl_return ElfCreateSymbolHandles( elf_sec_handle elf_sec_hnd )
{
    elf_sec_handle      sym_sec; // Nick's quickfix #02
        .       .       .
            default:
                // hotfix #02
```

```
if( ( sym_sec = ElfSymbolGetSecHandle( current ) ) != NULL
&& sym_sec->type == ORL_SEC_TYPE_PROG_BITS
&& sym_sec->flags & ORL_SEC_FLAG_EXEC
) {
    current->type = ORL_SYM_TYPE_FUNCTION;
} else {
    current->type = ORL_SYM_TYPE_NONE;
}
```

NOTE. This workaround works pretty well for current version, but has some drawbacks in “shared libraries” perspective. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than **STT_FUNC** will not be referenced automatically through the procedure linkage table.

The accurate fix should treat **STT_NOTYPE** as “normal” symbol.

4.1.3 Accurate segment mapping

Sections **.data** and **.bss** share the same segment in ELF executables produced by **wlink**. If **.bss** section is created, the memory size (**p_memsz**) of that segment became invalid. The produced ELF causes segmentation fault.

readelf -a

```
Section Headers:
        .       .       .
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf
Al
  [ 2] .text             PROGBITS        08048100 000100 00103d 00  AX  0   0
4
  [ 3] .data             PROGBITS        0804a000 002000 000288 00  WA  0   0
4
  [ 4] .bss              NOBITS          0804b000 003000 0000b4 00  WA  0   0
4
        .       .       .
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x00000000 0x00060 0x00060 R E 0
  LOAD           0x000100 0x08048100 0x00000000 0x0103d 0x0103d R E 0x1000
  LOAD           0x002000 0x0804a000 0x00000000 0x00288 0x000c4 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
```

```
   00
   01     .text
   02
```

It seems page alignment is not taken into account when **.bss** section is created.

QUICK FIX: **p_memsz** should be adjusted after creating the **.bss**.

```
$OWROOT/bld/wl/c/loadelf.c
            InitBSSSect( sh, off, CalcSplitSize(), linear );
            ph->p_memsz += ROUND_UP( ph->p_filesz, FmtData.objalign ); //
quickfix #03

readelf -a
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf
Al
        .       .       .
  [ 2] .text             PROGBITS        08048100 000100 00103d 00  AX  0   0
4
  [ 3] .data             PROGBITS        0804a000 002000 000288 00  WA  0   0
4
  [ 4] .bss              NOBITS          0804b000 003000 0000b4 00  WA  0   0
4
        .       .       .
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x00000000 0x00060 0x00060 R E 0
  LOAD           0x000100 0x08048100 0x00000000 0x0103d 0x0103d R E 0x1000
  LOAD           0x002000 0x0804a000 0x00000000 0x00288 0x010c4 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
   02     .data .bss
```

NOTE. This workaround works well enough, but the problem should be revised. More accurate fix is required.

5. Estimation

There are two independent tasks: Code Generator (PIC support) and Linker (building and using shared objects). So it is possible to perform these tasks simultaneously. GNU C Compiler can be used for Linker testing, as well as `ld` for Code Generator. Then the final integration (i.e. testing) should be performed, using only Open Watcom tools.

5.1 Position-Independent Code

Command line processing

Estimation: 1 day

Description: see section 3.1.1

Extending OWL

Estimation: 8 days

Description: see section 3.1.2

Implementing ELF output in CG386

Estimation: 20 days

Description: see section 3.1.2. This is pretty complicated task. No PIC support yet (see below).

Adding PIC support to CG386

Estimation: 20 days

Description: see section 3.1.3. This is one of most complicated tasks.

Extending ELF output in CG386

Estimation: 5 days

Description: see sections 3.1.2, 3.1.3. This task means adding PIC features to ELF.

Integration

Estimation: 5 days

Description: Mostly testing. Complicated test kit (i.e. C source code) is needed to ensure all things are implemented correctly.

Total **59 days**

5.2 Building Shared Objects

Extending ORL

Estimation: 3 days

Description: see section 3.2.7

Extending WLCore

Estimation: 15 days

Description: see sections 3.2.7, 3.2.8, 3.2.9

Improving LoadELF

Estimation: 5 days

Description: see sections 3.2.1 – 3.2.9

Integration

Estimation: 4 days

Description: Mostly testing. Complicated test kit (i.e. object files) is needed to ensure all things are implemented correctly.

Total **27 days**

5.3 Using Shared Objects

Command line processing

Estimation: 1 day

Description: see section 3.3.3

Improving LoadELF

Estimation: 1 day. Minimal changes are needed (assuming we are already able to build a shared object).

Description: see sections 3.3.2, 3.3.3

Extending ORL

Estimation: 10 days

Description: see section 3.3.1

Extending WLCore

Estimation: 8 days

Description: see sections 3.3.4, 3.3.5

Integration

Estimation: 10 days

Description: Mostly testing. Complicated test kit (i.e. object files) is needed to ensure all things are implemented correctly. There are many variants, e.g. executable uses three shared objects, where the 1st shared object uses the 2nd, and the 3rd uses some another shared object.

Total **30 days**

5.4 Final Integration

Estimation: 10 days

Description: Mostly testing. Trying to link ELF object files (i.e. those generated by Open Watcom C) using Open Watcom Linker.