Technical Documentation for the Revised ASxxxx Linker

Dr. Thomas Tensi

2009-10-08

1 Introduction

1.1 Overview

This program is a completely renovated version of the linker from the 8-bit-assembler suite ASxxxx from Alan Baldwin. Parts of this documentation are based on his suite documentation [Baldwin09].

The main idea was to modularize the program towards a plugin-architecture. All target platforms supported provide several routines which are called by the linking framework. Platform specific code should only go into those modules; platform-independent code should go into the general framework.

Apart from some syntactical improvements and data structure enhancements the main contribution of this linker is the support for bankswitching at link time. The SDCC compiler has provided bankswitching support for several platforms at compile time, but this proves to be too unflexible in practice. Take a library module for example: it may go to several banks depending on the bank layout of the calling program.

The renovated linker uses a configuration file which tells the bank number for each module. The linker puts the modules in the appropriate banks and also takes care that interbank calls are correctly handled.

2 Usage

The linker is a command line program with the following syntax and options:

```
aslink [-i] [-s] [-m] [-u] [-x] [-d] [-q] [-b area = value]
       [-g symbol = value] [-k path] [-l file]
       file ...
```

2.1 Platform Independent Options

The following options are supported:

file ...	list of files to be linked

General Options:

-b area = expression	specifies an area base address via an expression which may contain constants and/or defined symbols from the linked files (not yet supported)
-g symbol = expression	specifies value for the symbol via an expression which may contain constants and/or defined symbols from the linked files

Library File Options:

-k path	specifies a library directory prefix path; this will later be combined with a relative library path to find complete library path names
-l file	specifies the relative or complete path of a library file; when a relative path is given, the absolute path is found by combination with some library directory prefix path

Note that more than one path and also multiple library files are allowed.

Output Format Options:

-i	linker output is in Intel hex format (to `file.ihx`)
-s	linker output is in Motorola S19 format (to `file.s19`)

Note that output format options are cumulative; it is possible to have the output in more than one format.

Map and List File Options:

-m	generate a map file into `file.map` containing a list of symbols (grouped by area) with relocated addresses, sizes of linked areas, and other linking data
-u	update all available listing files `file.lst` into `file.rst` by replacing relocatable bytes by their correct absolute values
-x	use a hexadecimal number radix for the map file
-d	use a decimal number radix for the map file
-q	use an octal number radix for the map file

2.2 Specific Options for Z80/GBZ80 Platform

-j	generate a map file into `file.sym` in a form suitable for the debugger within the NO$GMB Gameboy emulator
-yo number	set count of ROM banks to `number` (default is 2)
-ya number	set count of RAM banks to `number` (default is 0)
-yt number	set cartridge MBC type to `number` (default is no MBC)
-yn name	set name of program to `name` (default is the name of output file truncated to eight characters)
-yp addr = byte	set byte in the output executable file at address `addr` to `byte`
-z	produce a Gameboy image as file (with extension `.gb`)

3 Processing

The linker gathers all command line files and processes them in the order presented in two passes.

In the first pass all modules, areas and symbols definitions and references are collected. For each referenced but undefined symbol all library files are searched and — when they contain such a symbol — are added to the object file list. This process continues until no more references can be satisfied. The library files are found by doing a combination of prefix paths and relative paths of the library files.

Finally area and symbol address definitions from the commandline are added to the internal symbol table.

When banking is used another step has to happen in pass 1: the banking module of the linker checks for *interbank calls*. Those interbank calls go from code in one bank to code in another parallel bank. Those calls cannot happen directly,

because during the call the banks have to be switched and also the bank switch upon return from the callee has to be organized. For those interbank calls the linker introduces trampoline code which does the above steps. Because banking also needs library modules another search of the libraries has to be done.

Note that at the end of the first pass it is clear what object files and libraries are needed for the executable and which symbols are available. It is not yet clear where the areas and symbols are located.

In the second pass all areas are located in the address space. This location is done based on the area type which is one of ABS, CON, REL, OVR or PAG as follows:

- Absolute areas (ABS) have a specific address either assembled in or given by the `-b` option on the linker command line. They are always put to the address given and are not relocated.
- Relative areas (REL) have a base address of `0x0000` assembled in. The real start address is either given by a `-b` option on the linker command line or is defined for the platform.

  All subsequent relative areas are concatenated to proceeding relative areas in an order defined by the first linker input file.

When all areas have been located, their addresses and those of the contained symbols have been completely defined.

Several checks are done for sanity of the output executable: The linker e.g. checks whether paged areas (type PAG) are on a 256 byte boundary and have a length less than or equal to 256 bytes. Also referenced but undefined symbols or inconsistent constants' definitions are reported.

As the main result of linking the output executable is now generated. The executable format is either one of the standard formats HEX or S-records or even a platform-specific format (like the Gameboy load format). The linker can even produce several output formats in parallel.

In addition to the output executables, two more kinds of output can be produced:

- A *linking map file* provides detailed information about symbol addresses, areas, modules, libraries and errors during linking.
- The *updated listing files* are generated from original assembler listings (.lst-files). For each of those files a companion file is produced which has all addresses and data relocated to their final values.

4 Banking Caveats

As mentioned before the linker takes care of transforming interbank-calls appropriately. Nevertheless there are some topics which cannot be correctly handled when an architecture does not use extended addresses for pointers. This, for example, applies to the GBZ80, which only has 16-bit pointers:

- an indirect call via a function pointer to a function in another bank, and
- pointer parameters pointing to another bank.

Note that the latter problem can be hard to track down: Take as an example a constant string (which is normally located in the bank of its compilation unit) and have its pointer passed to some routine as a parameter. When that routine is located in another bank or passes that parameter directly or indirectly to a routine in another bank, the access to the string will fail.

There are two remedies for that: either tag those constant strings as nonbanked — which is non-standard and uses up space in the nonbanked area — or copy those constants to nonbanked RAM – which is tedious and redundant —. Unfortunately in all cases the fact that banking occurs is visible to the programmer.

5 Architecture Overview

Similar to the original structure of Alan Baldwin's program the revised linker is organized in several modules according to the building blocks of the object files.

In the revised linker I have tried to abstract as much as possible from concrete underlying implementation structures (like linked lists) but provide access to them only via defined module interfaces. This makes the access sometimes a bit clumsier but encapsulates design decisions and also allows to optimize data structures centrally.

The top level module is the *Main* module which initializes and finalizes all other modules. It scans the commandline and gives all files found to the parser for a two-pass analysis. After the first pass undefined symbol references are resolved via the *Library* module.

The *Parser* module knows about the syntax of object and library files and calls the appropriate build routines from other modules in the first pass for symbol allocation and general construction of the internal network of related objects.

When banking is used, all interbank calls are modified using stub symbols during the first pass (where resulting trampoline calls and symbol definitions go into a synthetic object file).

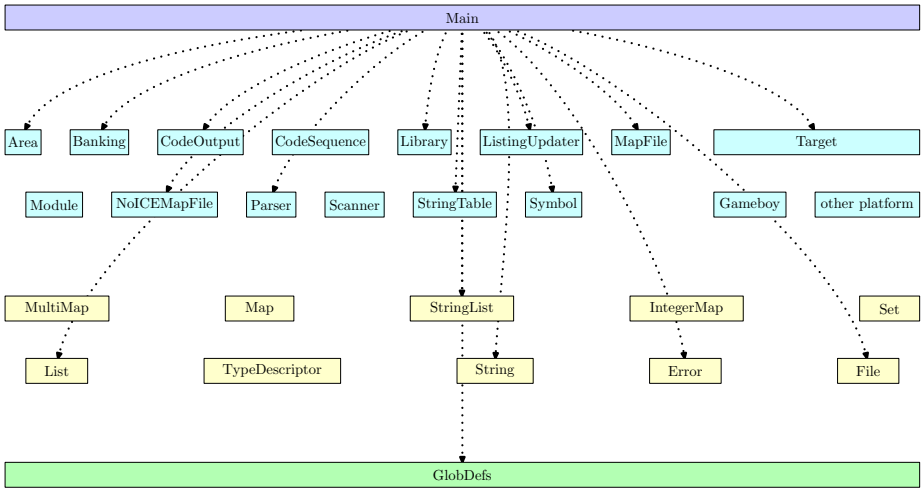

Figure 1: Module Dependencies of "Main" Module

At the end of the first pass the map file is written (if requested) using the *MapFile* module.

In the second pass the code is relocated by routines of the *CodeSequence* module. This module feeds the *CodeOutput* module which builds some internal representations for output of code in several formats. Note that it is possible to have several formats built in parallel (like e.g. S-records and Gameboy format).

At the end of the second pass — if requested — the *ListingUpdater* module traverses all available assembler listings and inserts the code data gathered by linking. This module is not very clever and heavily relies on the structure of the listing files currently provided by the SDCC compiler.

Figure 1 shows all modules of the linker and the usage dependencies of main. In the course of the document other usage dependencies will be illustrated where red arrow lines show dependencies from interface to another interface and black arrow lines show dependencies from implementation to another interface.

6 Module Interfaces

This section defines the module interfaces in detail. It is directly generated from the module include files leaving out implementation details exposed there.

All those interfaces follow an object-based approach. Whenever a module implements a type, there are routines to make instances of that type and those which manipulate object of that type (the latter have an object of the module type as a first parameter).

The naming convention used is as follows:

- All names have the module name as a prefix with single underscore for externally visible names and two underscores for internal names with file scope.
- The main type of a module is named `Type` (like e.g. `List_Type`).
- The constructor routines are called `make` or `makeXXX`, destructor routines `destroy`.
- When a module defines an object type with reference-semantics, those references point to private structures with some magic number. An object pointer can then be checked for validity by a routine `isValid`.
- If some internal module data must be kept, a module exports routines called `initialize` and `finalize` for initial setup and final cleanup. When initialization is done, also finalization must be executed, but a stateless module may have no such routines.

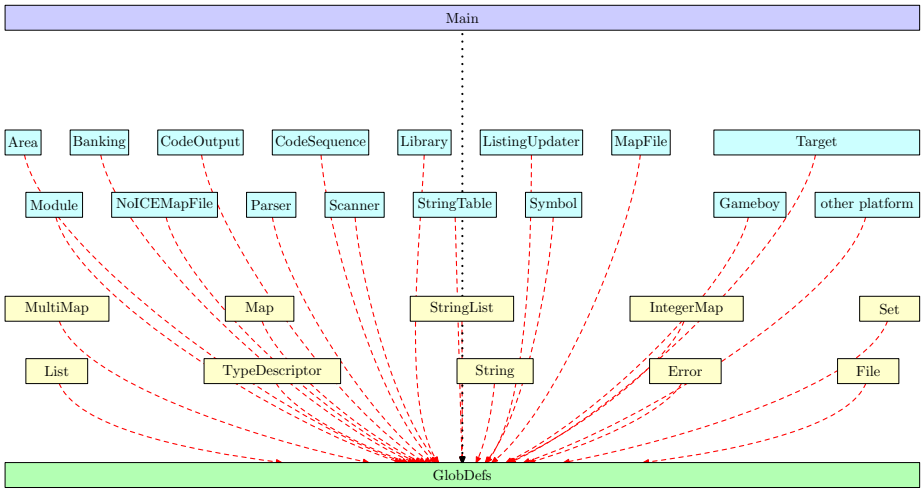

Figure 2: Module Dependencies onto "Globdefs" Module

6.1 Global Definitions Module

The module *GlobDefs* provides elementary types and routines used globally in the program. E.g., a `Boolean` type is defined here or the code templates for a very simple assertion checking. Because it is considered to be a globally known module, its identifiers are not prefixed with the module name (in contrast to the convention given above), but they are considered universal.

Figure 2 shows that all modules use the GlobDefs module.

6.1.1 Module "GlobDefs"

This module provides elementary types and routines for the generic ASXXX linker.

The types defined are several integer types, boolean and a generic object type. Additionally the module provides routines for some rudimentary form of assertion checking.

Because this module is considered to be globally known, its identifiers are universal and are not prefixed by the module name (contrary to the identifier prefix convention).

include <stddef.h>
include <stdlib.h>

typedef unsigned char UINT8;

typedef signed char INT8;

typedef unsigned int UINT16;

typedef signed int INT16;

typedef unsigned long UINT32;

typedef signed long INT32;

typedef size_t SizeType;

define SizeType_max ...

typedef int Boolean;

define false ...
define true ...

typedef void ∗Object;
placeholder type (representing all kinds of object types)

define in ...
define inout ...
define out ...
formal parameter modes (purely for documentation)

define NEW(elementType) ...
allocation routine for an object of some element type

define NEWARRAY(elementType, count) ...
allocation routine for an array of elements

define DESTROY(pointer) ...
deallocation routine for a pointer

Boolean PRE (**in** Boolean condition, **in char** ∗procName, **in char** ∗message);

checks precondition `condition`*; if false,* `message` *referring to* `procName` *is put out and program is terminated*

void ASSERTION (**in** Boolean condition, **in char** ∗procName, **in char** ∗message);

checks internal assertion `condition`*; if false,* `message` *referring to* `procName` *is put out and program is terminated*

Object attemptConversion (**in char** ∗opaqueTypeName, **in** Object object,
in long magicNumber);

generic routine for verifying that `object` *is a pointer to an internal type by checking the long integer pointed to; if it is not identical to* `magicNumber`*, the program stops with an error message*

Boolean isValidObject (**in** Object object, **in long** magicNumber);

generic routine for verifying that `object` *is a pointer to an internal type by checking the long integer pointed to; if it is not identical to* `magicNumber`*, the routine returns false*

6.2 Base Modules

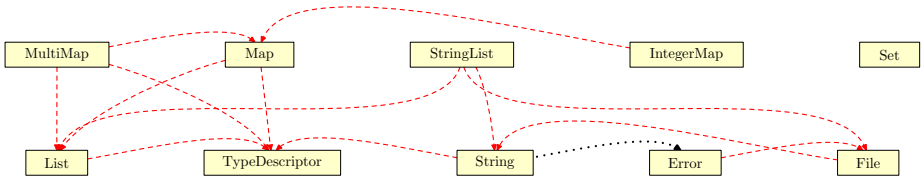

Figure 3: Module Dependencies between Base Modules

Those modules provide elementary services (like e.g. assertion checking) or base collection types (like e.g. lists).

- The *Error* module offers services for dealing with errors which are classified into several kinds of criticality and may lead to program abortion in case of fatal errors. Error output normally goes to stderr, but may be redirected to any open output file.
- The *File* module offers very simple file routines like opening, closing, reading and writing a file. For convenience several formatting routines are provided for numerals, strings and C-strings. The printf-style of C is intentionally not supported.

  For handling embedded information within a file a special convention is introduced: when a file name before the extension ends with '@' and a decimal number n this tells that the information starts at seek position n. In the linker this will be needed for object file libraries.

- The module *Set* is for set operations and it is implemented based on bitmaps. It can only handle sets of few integer elements between 0 and 31, but this is okay for our purposes. It has a value-based semantics so sets can be copied or passed as parameters like scalar types.
- The *String* module encapsulates services for strings. Strings are implemented as pointers to dynamically sized memory areas and those strings can be created, copied, filled from a C-string, searched, subscripted and so on. Note that strings have a reference-based semantics: directly assigning a string to another leads to an alias. Therefore the module has an explicit assignment operation which should be used whenever a copy of a string is needed. The *StringList* module only provides construction and printout of a list of strings; all other services can be taken from the `List` module.
- The *TypeDescriptor* module is needed by all generic collections (like lists and maps). A *type descriptor* is a structure that tells how some element in the container is constructed, destroyed, assigned, identified by some key and hashed. For normal scalar types those operations are trivial, for reference-based types some function pointers have to be stored in the structure. Whenever such an element is used in a collection container, the container knows how to act when some container element is created, destroyed and so on. When a routine pointer is `NULL`, nothing has to be done. A special default descriptor defined in the module consists only of NULL pointers and can be used for scalar types (which require no special handling).
- The *List* module provides services for generic lists. Elements can be concatenated to those lists, removed from them, searched for, etc.

  For iteration over a list a *cursor* can be used: it is set on the beginning of a list and gives some current element. It can be advanced and finally returns `NULL` as a signal that no more elements exist.

  A type descriptor for the embedded elements is given upon construction of a list. Consider e.g. a list of strings: when a string element is deleted from the list, the list should also free the resources of the string to prevent memory leaks.
- The *Map* module encapsulates services for generic maps, i.e. partial functions from keys of some type to values of some type. Those maps have at most one value for some key. As with lists above a type descriptor is needed for keys. The values in the maps are always considered as non-unique references and require no specific actions e.g. when a key-value-pair is deleted.

  IntegerMap is a variant of a map where the values are arbitrary integers. This implementation variant is necessary to allow 0 as a value.
- The *Multimap* module provides generic multimaps, i.e. partial functions from keys of some type to sets of values of some type (you can also interpret

them as relations between the two types grouped by the first relation partner). Similar to maps the values are considered reference-based, the key type is specified by a type descriptor upon multimap construction.

6.2.1 Module "Error"

This module provides all services for dealing with errors. Errors are classified into several kinds of criticality which define whether only an informational message is written or the program has to be stopped immediately because of a fatal situation.

Error output normally goes to stderr, but may be redirected to any open output file.

include "file.h"
include "globdefs.h"

```
typedef enum {
  Error_Criticality_warning,   Error_Criticality_error,
  Error_Criticality_fatalError
} Error_Criticality;
```

void Error_initialize (**void**);
sets up internal data structures

void Error_finalize (**void**);
cleans up internal data structures

void Error_setReportingTarget (**in** File_Type reportingFile);
all subsequent error output is directed to `reportingFile`

void Error_raise (**in** Error_Criticality criticality, **in char** ∗message, ...);
raises an error with `criticality` *displaying* `message`*; the message string may be a printf-style template with parameters which are filled by the optional arguments*

6.2.2 Module "File"

This module provides all services for handling files in the generic SDCC linker.

A file is specified by a file name which is a string in a platform specific notation. The path separator is a variable that is set according to the local convention for path separation (a slash in Unix and a backslash in Windows).

When a file name ends in an at-character followed by a decimal number, its relevant information is considered to start at that offset. This means that 'file' and 'file@0' mean the same.

A file may be opened in several modes where read and write as well as binary and text variants are distinguished. Writing to a file means that all previous contents starting at the write position are discarded.

Because we are only dealing with text files as input to the linker, there is only one read routine which returns a single line from a file (terminated by a newline string). The write routines are type-specific; there is intentionally no explicit printf-style routine. Nevertheless for use by other modules there is also a vararg write routine.

After processing a file must be explicitely closed.

based on the module lkfile.c by Alan R. Baldwin

include <stdarg.h>

include "globdefs.h"
include "string.h"

typedef struct File__Record *File_Type;
type representing a file

typedef enum {
File_Mode_read, File_Mode_write, File_Mode_readBinary, File_Mode_writeBinary
} File_Mode;
open mode for a file; binary and text modes are distinguished

extern File_Type File_stderr;
standard error file stream, typically routed to the console

extern String_Type File_directorySeparator;
string to separate parts of a directory specification; stands for "/" in Unix and "in Windows

define File_offsetSeparator ...
character to separate offset part of a file name from the plain file name

void File_initialize (**void**);
sets up internal data structures for this module

void File_finalize (**void**);
cleans up internal data structures for this module

Boolean File_open (**out** File_Type *file,
in String_Type fileName, **in** File_Mode mode);
opens file given by `fileName` for reading or writing depending on `mode`; if successful, `file` contains the associated file, otherwise false is returned; note that the system supports the special file names "stdin", "stdout" and "stderr" which access the appropriate terminal streams; when a file is opened for writing, its previous contents are discarded (possibly in between when the file name contains an offset separator)

void File_close (**inout** File_Type *file);
ends processing of `file`

Boolean File_exists (**in** String_Type fileName);

tells whether file given by `fileName` *exists*

void File_readLine (**inout** File_Type *file, **out** String_Type *st);
returns next line on `file` *in* `st` *including a final newline character; when file is exhausted,* `st` *is empty*

void File_writeBytes (**inout** File_Type *file, **in** UINT8 *data, **in** SizeType size);
puts byte array `data` *with length* `size` *to* `file`

void File_writeChar (**inout** File_Type *file, **in char** ch);
puts character `ch` *to* `file`

void File_writeCharArray (**inout** File_Type *file, **in char** *st);
puts NUL-terminated character array `st` *to* `file`

void File_writeHex (**inout** File_Type *file,
in UINT32 value, **in** UINT8 digitCount);
puts `digitCount` *least significant bytes of* `value` *in hexadecimal to* `file`

void File_writePrintfArguments (**inout** File_Type *file, **in char** *format,
in StdArg_VarArgList argumentList);
puts all arguments in `argumentList` *according to printf-style* `format` *to* `file`

void File_writeString (**inout** File_Type *file, **in** String_Type st);
puts string `st` *to* `file`

6.2.3 Module "IntegerMap"

This module provides all services for maps from objects to long integers. It redefines only the `set` *and* `lookup` *routine; the other* `Map` *routines may be used as is.*

This wrapper module is necessary because the map module uses pointers as map values and returns `NULL` *as a failure indicator. When an integer 0 occurs as a map value, it hence cannot be distinguished from a* `NULL` *pointer by the* `Map` *module and cannot be used.* `IntegerMap` *does some internal bookkeeping and hides this complication from its clients which can safely use 0 as a value in an integer map. Instead of* `NULL` *it returns* `notFound` *when a lookup fails.*

include "globdefs.h"
include "map.h"

typedef Map_Type IntegerMap_Type;
redefined integer map type based on generic map type

define IntegerMap_notFound ...
value returned when lookup fails

long IntegerMap_lookup (**in** IntegerMap_Type map, **in** Object key);

searches `map` for element with identification `key` and returns associated value or `notFound` if none exists

void IntegerMap_set (**inout** IntegerMap_Type *map, **in** Object key, **in long** value);
sets `value` for `key` in `map`

6.2.4 Module "List"

This module provides all services for generic lists. A generic list can take elements of fixed size where the size is specified upon list creation. Additionally elements can be searched in the list and appended to the list.

For iteration a cursor can be defined on a list and is used to linearly traverse it and inspect, change or delete the element in the list where it points to.

A type descriptor for the embedded elements is given upon construction of a list. This is necessary e.g. for additional cleanup or allocation operations when elements are deleted or created. For scalar or purely reference-based types the default type descriptor is okay, because those types do not need any additional processing in those cases.

include "globdefs.h"
include "typedescriptor.h"

extern TypeDescriptor_Type List_typeDescriptor;
variable used for describing the type properties when list objects occur in generic types (like other lists)

typedef struct List__Record *List_Type;
generic list

typedef struct List__Linkable *List_Cursor;
cursor to some element within a list

void List_initialize (**void**);
sets up internal data structures for this module

void List_finalize (**void**);
cleans up internal data structures for this module

Boolean List_isValid (**in** Object list);
checks whether `list` is a valid list

TypeDescriptor_Type List_getElementType (**in** List_Type list);
returns the type descriptor for the elements in `list`

List_Type List_make (**in** TypeDescriptor_Type typeDescriptor);
constructs a single list with elements of the type specified in `typeDescriptor`; this parameter specifies element size and how single elements behave on construction, destruction and assignment and how they are compared

void List_destroy (**inout** List_Type *list);
destroys all elements in `list` *and list itself*

Object List_lookup (**in** List_Type list, **in** Object key);
searches `list` *for element with identification* `key` *and returns it or NULL if no such element exists*

Object List_getElement (**in** List_Type list, **in** SizeType i);
returns `i`*-th element of* `list`

SizeType List_length (**in** List_Type list);
returns the length of `list`

Object *List_append (**inout** List_Type *list);
appends newly allocated element to end of `list` *and returns a pointer this new list element*

void List_clear (**inout** List_Type *list);
removes all elements in `list`

void List_copy (**inout** List_Type *destination, **in** List_Type source);
copies contents from `source` *to* `destination`

void List_concatenate (**inout** List_Type *list, **in** List_Type otherList);
concatenates contents of `otherList` *to* `list`

List_Cursor List_resetCursor (**in** List_Type list);
returns cursor on head of `list`*; if list is empty, result is NULL*

List_Cursor List_setCursorToElement (**in** List_Type list, **in** Object key);

Object List_getElementAtCursor (**in** List_Cursor cursor);
gets element pointed at by `cursor`

void List_putElementToCursor (**in** List_Cursor cursor, **in** Object newValue);
assigns `newValue` *to element pointed at by* `cursor`

void List_deleteElementAtCursor (**in** List_Cursor cursor);

void List_advanceCursor (**inout** List_Cursor *cursor);
advances `cursor` *by one element within associated list; if list is exhausted,* `cursor` *is set to NULL*

6.2.5 Module "Map"

This module provides all services for generic maps. Those maps represent partial functions from keys to values and have at most one value for some key.

It is possible to add some value for a key, check the assigned values, remove some value or even delete the assignment for some key completely.

Because map entries are inserted, assigned to and discarded, a type descriptor for keys must be provided when a map is created. There is no such descriptor for the values in the maps, because they are always considered as non-unique references and require no specific action.

include "globdefs.h"
include "list.h"
include "typedescriptor.h"

typedef struct Map__Record *Map_Type;
map type based on private underlying structure type

void Map_initialize (**void**);
sets up internal data structures for this module

void Map_finalize (**void**);
cleans up internal data structures for this module

Boolean Map_isValid (**in** Object map);
checks whether `map` *is a valid map*

TypeDescriptor_Type Map_getKeyType (**in** Map_Type map);
gets type descriptor for keys in `map`

Map_Type Map_make (**in** TypeDescriptor_Type keyTypeDescriptor);
constructs a single map with keys conforming to `keyTypeDescriptor` *and object values*

void Map_destroy (**out** Map_Type *map);
destroys all elements in `map` *and map itself*

Object Map_lookup (**in** Map_Type map, **in** Object key);
searches `map` *for element with identification* `key` *and returns associated value or NULL if none exists*

void Map_getKeyList (**in** Map_Type map, **inout** List_Type *keyList);
gets list of keys of `map` *and returns them in* `keyList`

void Map_clear (**inout** Map_Type *map);
removes all elements in `map`

void Map_set (**inout** Map_Type *map, **in** Object key, **in** Object value);
sets `value` *for* `key` *in* `map`

void Map_deleteKey (**inout** Map_Type *map, **in** Object key);
removes `key` *from* `map`*; when* `key` *is not in* `map`*, nothing happens*

6.2.6 Module "Multimap"

This module provides all services for generic multimaps. Those multimaps represent partial functions from keys to sets of values and have at most one set of values for some key.

It is possible to add some value for a key, check the assigned values, remove some value or even delete the assignment for some key completely.

include "globdefs.h"
include "map.h"
include "typedescriptor.h"

typedef Map_Type Multimap_Type;
multimap type based on map type

void Multimap_initialize (**void**);
sets up internal data structures for this module

void Multimap_finalize (**void**);
cleans up internal data structures for this module

Boolean Multimap_isValid (**in** Object map);
checks whether `map` *is a valid multimap*

TypeDescriptor_Type Multimap_getKeyType (**in** Multimap_Type map);
gets type descriptor for keys in `map`

Multimap_Type Multimap_make (**in** TypeDescriptor_Type keyTypeDescriptor);
constructs a single multimap with keys conforming to `keyTypeDescriptor`

void Multimap_destroy (**out** Multimap_Type *map);
destroys all elements in `map` *and map itself*

List_Type Multimap_lookup (**in** Multimap_Type map, **in** Object key);
searches `map` *for element with identification* `key` *and returns associated value list or NULL if none exists*

void Multimap_clear (**inout** Multimap_Type *map);
removes all elements in `map`

void Multimap_add (**inout** Multimap_Type *map, **in** Object key, **in** Object value);
adds `value` *to* `key` *in* `map`

void Multimap_deleteKey (**inout** Multimap_Type *map, **in** Object key);
removes `key` *and all its entries from* `map`*; when* `key` *is not in* `map`*, nothing happens*

void Multimap_deleteValue (**inout** Multimap_Type *m,
in Object key, **in** Object value);

6.2.7 Module "Set"

This module provides all services for handling sets represented as long integers (bit-sets). Those sets can only carry integer elements in the range of 0 to 31 (or enumeration types).

The standard routines are available to make a set empty (`clear`), to check whether it is empty (`isEmpty`), to find some element in the set (`firstElement`), to make a singleton set from one element (`make`), to find the complement set (`complement`), to test set membership (`isElement`) and to in-/exclude an element (`include`, `exclude`).

include "globdefs.h"

typedef long Set_Type;
a set may contain the elements 0..31

typedef char Set_Element;
the base type of the set

void Set_initialize (**void**);
sets up internal data structures

void Set_finalize (**void**);
cleans up internal data structures

Set_Type Set_make (**in** Set_Element element);
makes a singleton set from `element`

void Set_clear (**out** Set_Type *set);
makes `set` empty

Set_Element Set_firstElement (**in** Set_Type set);
finds out first element in `set` when not empty

Boolean Set_isElement (**in** Set_Type set, **in** Set_Element element);
tells whether `element` occurs in `set` or not

Boolean Set_isEmpty (**in** Set_Type set);
finds out whether `set` is empty

Set_Type Set_complement (**in** Set_Type set);
returns the complement set of `set`

void Set_include (**inout** Set_Type *set, **in** Set_Element element);
adds `element` to `set`

void Set_exclude (**inout** Set_Type *set, **in** Set_Element element);
removes `element` from `set`

6.2.8 Module "String"

This module provides all services for handling strings in the SDCC linker.

Those strings are pointers to dynamically sized memory areas and are not type compatible with C-strings. There is a complete set of manipulation routines available (assignment, search, slicing, ...) and also the conversion from and to integers or C-strings.

Note that the first character in a string has index 1.

include "globdefs.h"
include "typedescriptor.h"

define String_terminator ...
character defining the end of a character array

define String_notFound ...
value returned when some string lookup routine fails

typedef struct String__Record *String_Type;
a character string

extern TypeDescriptor_Type String_typeDescriptor;
variable used for describing the type properties when string objects occur in generic types like lists

extern String_Type String_newline;
string representing a newline

extern String_Type String_emptyString;
an empty string

void String_initialize (**void**);
sets up internal data structures for this module

void String_finalize (**void**);
cleans up internal data structures for this module

String_Type String_allocate (**in** SizeType capacity);
allocates string with at most `capacity` *significant characters*

String_Type String_make (**void**);
constructs empty string

String_Type String_makeFromCharArray (**in char** *source);
constructs string from character array `source`

void String_destroy (**inout** String_Type *st);
deallocates string `st`

char String_getAt (**in** String_Type st, **in** SizeType i);

gets character at `i`-th position in `st` and returns it; when `i` is less than 1 or greater than the string length, this routine fails

char *String_asCharPointer (**in** String_Type st);

returns character array representation of string terminated by NUL

void String_clear (**inout** String_Type *st);

clears contents of `st`

void String_copy (**out** String_Type *destination, **in** String_Type source);

copies contents of `source` into `destination`

void String_copyAligned (**out** String_Type *destination,
in UINT8 maxLength, **in** String_Type source,
in char fillChar, **in** Boolean isLeftAligned);

formats `source` into `destination` using at most `maxLength` characters; if `source` is shorter than `maxLength`, the remaining space is filled with `fillChar`; when `isLeftAligned` source is aligned left in destination, otherwise right

void String_copyCharArray (**out** String_Type *destination, **in char** *source);

copies contents of `source` into `destination` up to and including `String_terminator`

void String_copyCharArrayAligned (**out** String_Type *destination,
in UINT8 maxLength, **in char** *source,
in char fillChar, **in** Boolean isLeftAligned);

formats `source` into `destination` using at most `maxLength` characters; if `source` is shorter than `maxLength`, the remaining space is filled with `fillChar`; when `isLeftAligned` source is aligned left in destination, otherwise right

void String_copyInteger (**out** String_Type *destination, **in** INT32 value,
in UINT8 base);

formats `value` with `base` and copies result into `destination`

void String_copyIntegerAligned (**out** String_Type *destination,
in UINT8 maxLength, **in** INT32 value,
in UINT8 base,
in char fillChar, **in** Boolean isLeftAligned);

formats integer `value` with `base` into `destination` using at most `maxLength` characters; if resulting number string is shorter than `maxLength`, the remaining space is filled with `fillChar`; when `isLeftAligned` source is aligned left in destination, otherwise right

void String_append (**inout** String_Type *destination,
in String_Type otherString);

appends contents of `otherString` to `destination`

void String_appendChar (**inout** String_Type *destination, **in char** ch);

appends character `ch` to `destination`

void String_appendCharArray (**inout** String_Type *destination,
in char *otherString);

appends contents of `otherString` *to* `destination`

void String_appendInteger (**out** String_Type *destination, **in** UINT32 value,
in UINT8 base);

formats `value` *with base* `base` *and appends result to* `destination`

void String_deleteCharacters (**inout** String_Type *st, **in** SizeType position,
in SizeType count);

deletes `count` *characters in* `st` *starting at* `position`

void String_fillWithCharacter (**inout** String_Type *st, **in char** ch,
in SizeType count);

fills first `count` *characters of* `st` *with character* `ch`

void String_prepend (**inout** String_Type *destination,
in String_Type otherString);

prepends contents of `otherString` *to* `destination`

void String_prependChar (**inout** String_Type *destination, **in char** ch);

prepends character `ch` *to* `destination`

void String_prependCharArray (**inout** String_Type *destination,
in char *otherString);

prepends contents of `otherString` *to* `destination`

void String_prependInteger (**out** String_Type *destination, **in** UINT32 value,
in UINT8 base);

formats `value` *with base* `base` *and prepends result to* `destination`

void String_removeTrailingCrLf (**inout** String_Type *st);

removes trailing line feed or carriage return characters of `st`

void String_convertToCharArray (**in** String_Type st, **in** SizeType maxSize,
out char *chList);

puts contents of `st` *into character array* `chList` *terminated by* `String_terminator` *when length of* `st` *is less or equal to* `maxSize`

Boolean String_convertToLong (**in** String_Type st, **in** UINT8 defaultBase,
out long *result);

parses contents of `st` *as long number with default base* `defaultBase` *and returns result in* `result`*; any base changing prefices (like "0x") are interpreted; returns false on failure*

void String_convertToUpperCase (**in** String_Type st, **out** String_Type *result);

returns upper case representation of `st` *in* `result`

SizeType String_findCharacter (**in** String_Type st, **in char** ch);

locates `ch` *in* `st` *and returns its position; when* `ch` *does not occur,* `notFound` *is returned*

SizeType String_findCharacterFromEnd (**in** String_Type st, **in char** ch);
locates `ch` *in* `st` *starting at end of string and returns its position; when* `ch` *does not occur,* `notFound` *is returned*

SizeType String_find (**in** String_Type st, **in** String_Type substring);
locates `substring` *in* `st` *and returns its position; when* `substring` *does not occur,* `notFound` *is returned*

SizeType String_findFromEnd (**in** String_Type st, **in** String_Type substring);
locates `substring` *in* `st` *starting at end of string and returns its position; when* `substring` *does not occur,* `notFound` *is returned*

char String_getCharacter (**in** String_Type st, **in** SizeType i);
gets `i`*-th character in* `st` *where the first character has index 1*

void String_getSubstring (**out** String_Type ∗result, **in** String_Type st,
in SizeType startPosition, **in** SizeType count);
gets substring of `st` *from* `startPosition` *of at most* `count` *characters into* `result`*; fails when* `startPosition` *is non-positive or* `count` *is negative*

Boolean String_hasPrefix (**in** String_Type st, **in** String_Type prefix);
tells whether `st` *has leading* `prefix`

Boolean String_hasSuffix (**in** String_Type st, **in** String_Type suffix);
tells whether `st` *has trailing* `suffix`

SizeType String_length (**in** String_Type st);
returns length of string `st`

Boolean String_isEqual (**in** String_Type strA, **in** String_Type strB);
tells whether two strings `strA` *and* `strB` *are equal*

SizeType String_hashCode (**in** String_Type st);
computes the hash code for string `st`*; a simple hash code is used: if the sequence of character codes is c_1,c_2,...,c_n, its hash value will be* $SUM\_i(2^{n-i} * c\_i)$

6.2.9 Module “StringList”

This module provides services for lists of strings.

It only provides a specific constructor, a customized append routine and a write routine. For all other operations the standard list routines must be used.

include ”file.h”
include ”list.h”
include ”string.h”

typedef List_Type StringList_Type;

StringList_Type StringList_make (**void**);
constructs a single list with string elements

void StringList_append (**inout** StringList_Type *list, **in** String_Type st);
appends string `st` *to end of* `list`

void StringList_write (**in** StringList_Type list, **inout** File_Type *file,
in String_Type separator);
writes `list` *to* `file`*, where all entries are terminated by* `separator`

6.2.10 Module "TypeDescriptor"

This module encapsulates services for generic handling of objects within some container structures. Containers within the linker always use pointers to elements. The elementary operations for those element types must be defined and are referenced by a type descriptor.

Those elementary operations are: construction of some object, destruction of an object, assignment of one object to another, comparison of two objects for equality, calculating some hash code for an object and checking whether some object has a specific key. When such a routine is not defined for some type, a reasonable default (like e.g. a pointer comparison) is used.

include "globdefs.h"

typedef void (*TypeDescriptor_AssignmentProc)(**in** Object *destination,
in Object source);
generic routine type describing the assignment of an object to another

typedef Boolean (*TypeDescriptor_ComparisonProc)(**in** Object objectA,
in Object objectB);
generic routine type describing the equality check of two objects

typedef Object (*TypeDescriptor_ConstructionProc)(**void**);
generic routine type describing the creation of an object

typedef void (*TypeDescriptor_DestructionProc)(**inout** Object *object);
generic routine type describing the destrucion of an object

typedef SizeType (*TypeDescriptor_HashCodeProc)(**in** Object object);
generic routine type describing the hash code calculation for an object

typedef Boolean (*TypeDescriptor_KeyValidationProc)(**in** Object object,
in Object key);
generic routine type describing the check of some object against some key

typedef struct {
  SizeType objectSize;
  TypeDescriptor_AssignmentProc assignmentProc;
  TypeDescriptor_ComparisonProc comparisonProc;
  TypeDescriptor_ConstructionProc constructionProc;
  TypeDescriptor_DestructionProc destructionProc;
  TypeDescriptor_HashCodeProc hashCodeProc;
  TypeDescriptor_KeyValidationProc keyValidationProc;
} TypeDescriptor_Record;

type defining the central characteristics of some type: its size, routines for assignment, construction, destruction, comparison and key validation; when a routine is NULL, the corresponding bitwise operations are used as a default (e.g. a memmove for assignment)

typedef TypeDescriptor_Record *TypeDescriptor_Type;

extern TypeDescriptor_Type TypeDescriptor_default;

default type descriptor using standard bitwise operations; this descriptor can e.g. be used for containers of plain pointers which have no constructors, destructors or specific assignment operations

extern TypeDescriptor_Type TypeDescriptor_plainDataTypeDescriptor;

type descriptor for plain data types (like integers) using standard bitwise operations and a hash code which interpretes the value of the data type

void TypeDescriptor_initialize (**void**);

sets up internal data structures for this module

void TypeDescriptor_finalize (**void**);

cleans up internal data structures for this module

void TypeDescriptor_assignObject (**in** TypeDescriptor_Type typeDescriptor,
inout Object *destination,
in Object source);

assigns `source` *to* `destination` *with the assignment routine defined in* `typeDescriptor`

Boolean TypeDescriptor_checkObjectForKey (
in TypeDescriptor_Type typeDescriptor,
in Object object, **in** Object key);

checks whether `object` *has* `key` *with the key checking routine defined in* `typeDescriptor`

Boolean TypeDescriptor_compareObjects (
in TypeDescriptor_Type typeDescriptor,
in Object objectA, **in** Object objectB);

checks whether `objectA` *and* `objectB` *are equal with the equality checking routine defined in* `typeDescriptor`

void TypeDescriptor_destroyObject (**in** TypeDescriptor_Type typeDescriptor,
inout Object *object);

destroys existing `object` *with the destruction routine defined in* `typeDescriptor` *and nullifies pointer*

Object TypeDescriptor_makeObject (**in** TypeDescriptor_Type typeDescriptor);

makes new object with the creation routine defined in `typeDescriptor`

SizeType TypeDescriptor_objectHashCode (**in** TypeDescriptor_Type typeDescriptor,
in Object object);

returns hash code for `object` *with the hash code calculation routine defined in* `typeDescriptor`

6.3 Linker Specific Modules

Those modules provide linker-specific services and rely on the base modules defined in the previous section.

- The module *Area* handles areas and segments. Those are groups of code or data with similar properties like being overlayed or getting assigned to some specific memory location. Areas are abstract groupw while segments are concrete instances of areas within some object module.

  Normally there is some current area where all symbols and code encountered in a code module is put.

- The module *Banking* handles interbank calls by introducing trampoline calls in the nonbanked area. This requires some trickery by synthesizing an object file. The mechanism is completely platform-independent, because all specifics are delegated by callbacks to the Target module.

- The module *CodeOutput* centralizes the output to the code files. Several such output streams are available and they may use arbitrary output formats. Each stream has to be registered upon program startup and uses a platform-dependent code output routine. When some code sequence has to be put out, CodeOutput dispatches that request to all open output streams. An example of a platform-specific output routine can be found in the Gameboy module.

- The module *CodeSequence* encapsulates code sequences, i.e. byte sequences to be put to some bank at some specific address. One central routine can apply a list of simple relocations to a given code sequence, where a simple relocation specifies some position in the code sequence, an offset value and some indication on how the value pointed to has to be combined with the offset (e.g. added).

- The module *Library* encapsulates services for object file libraries. Those are searched for in directories and under specific names. A single routine searches all matching files for symbol definitions matching unresolved

symbol references at the end of the first linking pass, resolves those and adds the matching files to the code base and the symbols to the symbol table for further processing.

- The module *ListingUpdater* updates assembler listings associated with all link objects files (except for libraries) by inserting relocated code at appropriate places.
- The module *MapFile* provides services for generation of map files to give an overview about the object files read, the allocation of symbols and areas and the library files used.

  Several map files can be open at once and each may have a different routine for output. Those target files and map output routines must be registered at the MapFile module and are then automatically fed.
- The module *Module* encapsulates the concept of a "module". A module is a group of code and data areas belonging together and is the root of all related linker objects. Its associated areas and symbols are accessible via several routines.
- The module *Parser* is responsible for analysing tokenized character streams. Those can be single or list of object files where the parser calls other modules to build up internal object structures.

  A reduced scan for a simple list of symbols encountered is also possible and the analysis of key-value-assignments by equations.
- The module *scanner* gathers character streams to tokens. Those character streams are given as a read-character-routine and the tokens are — among others — identifier, numbers and operators. As typical for parsing, tokens may be pushed back onto input to allow for some lookahead during parsing.
- The module *stringtable* encapsulates two string tables (or string lists) containing the global base address definitions and the global symbol definitions as strings.
- The module *symbol* administers symbols with name, associated segment and address. The internal table also stored whether a symbol has been defined, referenced or both. To assist banking a symbol may be split into a real and a surrogate symbol and finally a list of referenced but undefined symbols may be obtained.

6.3.1 Module "Area"

This module provides all services for area definitions in the generic SDCC linker.

An area is a group of code or data snippets which share some properties. E.g., code within an area may be overlayed if the area specifies that. Areas are abstract groups

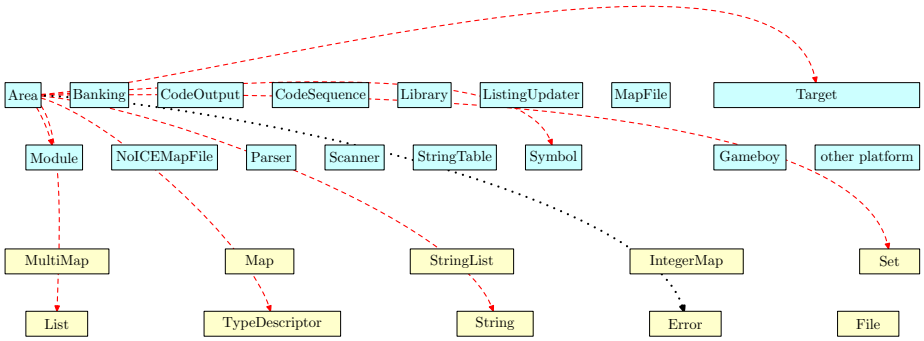

Figure 4: Module Dependencies of "Area" Module

that are defined by segments within modules. I.e., segments within object modules with the same name are considered to belong to the same area and inherit its attributes.

One can query this module for all areas, for the name and attributes of a specific area, the list of its segments and so on. It is also possible to set most of those properties as well.

There is also the concept of a current area, which is the area currently active during processing of a module. All definitions go to the segment belonging to this area and the current module.

Finally an area may be linked: all addresses of segments within that area are resolved (depending on whether they are overlayed or concatenated).

based on the module lkarea.c by Alan R. Baldwin

```
include "globdefs.h"
include "list.h"
include "set.h"
include "string.h"
include "target.h"
include "typedescriptor.h"
```

typedef struct Area__SegmentRecord *Area_Segment;
a segment within an area (as an opaque type)

typedef List_Type Area_SegmentList;
a list of segments

```
include "module.h"
include "symbol.h"
```

typedef struct Area__Record *Area_Type;
type representing a group of link segments

typedef enum {

Area_Attribute_isAbsolute, Area_Attribute_hasOverlayedSegments,
Area_Attribute_hasPagedSegments, Area_Attribute_isInCodeSpace,
Area_Attribute_isInExternalDataSpace, Area_Attribute_isInBitSpace,
Area_Attribute_isNonloadable
} Area_Attribute;

properties of an area like being absolute, having overlayed or paged segments and special flags for different other target platforms; absence of certain properties is specified by not including them into the set

typedef Set_Type Area_AttributeSet;

set of Area_Attribute

typedef List_Type Area_List;

a list of areas

extern TypeDescriptor_Type Area_typeDescriptor;

variable used for describing the type properties when area objects occur in generic types like lists

extern TypeDescriptor_Type Area_segmentTypeDescriptor;

variable used for describing the type properties when segment objects occur in generic types like lists

void Area_initialize (**void**);

sets up all internal data structures

void Area_finalize (**void**);

cleans up all internal data structures

Area_Type Area_make (**in** String_Type areaName,
in Area_AttributeSet attributeSet);

ensures that a new area with `areaName` *exists; its attributes are given by* `attributeSet`*; when set of attributes is not identical to previous area definition, this is an error*

void Area_makeSegment (**in** String_Type areaName,
in Target_Address totalSize,
in Area_AttributeSet attributeSet);

adds a new segment to area with `areaName`*; size of segment is given by* `totalSize`*, attributes by* `attributeSet`*; when set of attributes is not identical to previous area definition, this is an error*

void Area_makeAbsoluteSegment (**void**);

adds a new segment to absolute area

Area_AttributeSet Area_makeAttributeSet (**in** UINT8 attributeSetEncoding);

constructs an area attribute set from some external encoding

void Area_destroy (**inout** Area_Type *area);

destroys `area` *completely and frees all its resources*

Area_Segment Area_currentSegment (**void**);

returns currently active segment

void Area_getList (**inout** Area_List *areaList);
returns list of all known areas in `areaList`

void Area_getName (**in** Area_Type area, **out** String_Type *name);
returns name of `area` *in* `name`

Area_AttributeSet Area_getAttributes (**in** Area_Type area);
returns attributes of `area`

Target_Address Area_getAddress (**in** Area_Type area);
returns address of `area`

void Area_getListOfSegments (**in** Area_Type area,
out Area_SegmentList *segmentList);
returns segments of `area` *in* `segmentList`

UINT8 Area_getMemoryPage (**in** Area_Type area);
returns assigned memory page for `area` *(if applicable for platform)*

Target_Address Area_getSegmentAddress (**in** Area_Segment segment);
returns address of `segment`

Area_Type Area_getSegmentArea (**in** Area_Segment segment);
returns area of `segment`

void Area_getSegmentName (**in** Area_Segment segment, **out** String_Type *name);
returns name of `segment` *in* `name`

Module_Type Area_getSegmentModule (**in** Area_Segment segment);
returns associated module for `segment`

void Area_getSegmentSymbols (**in** Area_Segment segment,
out Symbol_List *symbolList);
returns all symbols in `segment` *in* `symbolList`

Target_Address Area_getSize (**in** Area_Type area);
returns size of `area`

void Area_setCurrent (**in** Area_Segment segment);
sets current segment to `segment`

void Area_lookup (**out** Area_Type *area, **in** String_Type areaName);
looks up area with `name` *and sets area accordingly (or to NULL, if not found)*

void Area_addSymbolToSegment (**inout** Area_Segment *segment,
in Symbol_Type symbol);
adds `symbol` *to module of* `segment`

void Area_clearListOfSegments (**inout** Area_Type *area);

removes all segments of `area`

void Area_link (**void**);

*resolves all area addresses by traversing all the areas and the associated segments; the address allocation is done depending on the attributes of the area: - for overlayed areas all segments starts at the identical base area address overlaying each other and the size of the area is the maximum of the area segments - for concatenated areas all segments are concatenated with the first segment starting at the base area address and the size of the area is the sum of the segment sizes if a base address for an area is specified then the area will start at that address. Any relocatable areas defined subsequently will be concatenated to the previous relocatable area if it does not have a base address specified; additionally the symbols named s_*`areaName` *and l_*`areaName` *are created to define the starting address and length of each area*

void Area_replaceSegmentSymbol (**inout** Area_Segment *segment,
in Symbol_Type oldSymbol,
in Symbol_Type newSymbol);

replaces `oldSymbol` *in symbol list of* `segment` *by* `newSymbol`*; does nothing when* `oldSymbol` *does not occur*

void Area_setBaseAddresses (**in** String_Type segmentName,
in Target_Address baseAddress);

sets addresses of all segments with `segmentName` *to* `baseAddress`

void Area_setSegmentArea (**inout** Area_Segment *segment, **in** Area_Type area);

sets area of `segment` *to* `area`

void Area_toString (**in** Area_Type area, **out** String_Type *representation);

constructs a printable representation of `area` *and its internal data (for debugging purposes) and concatenates it to* `representation`

void Area_segmentToString (**in** Area_Segment segment,
out String_Type *representation);

constructs a printable representation of `segment` *and its internal data (for debugging purposes) and concatenates it to* `representation`

6.3.2 Module "Banking"

This module provides all services for code banking in the generic linker.

Banking is done by reading a banking configuration file which tells the assignments of modules to bank numbers. Whenever a jump from one code bank to a parallel bank occurs, this jump is replaced by a jump to stub code in a nonbanked area which takes care of the bank switching and also restores the calling bank when a return occurs (for a subroutine call).

Unfortunately this mechanism only works for control flow via bank boundaries! It fails e.g. when some constant should be read from another code bank. So you should not

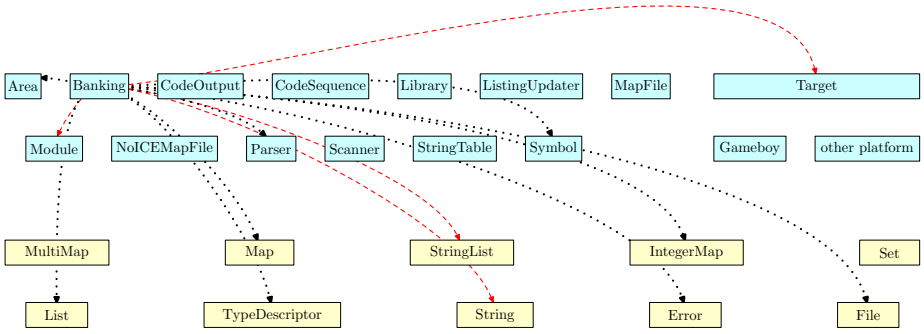

Figure 5: Module Dependencies of "Banking" Module

do it (also not accidentally by passing constants as parameters to routines in other banks)...

All this code change is done on-the-fly in the linker based on the configuration file. So the decision about bank assignment can be done at link time and must not be done at compile time. This also means that libraries can be arbitrarily placed in banks without recompilation.

After the first link pass when all symbols have been resolved, the linker tracks whether interbank references occur. For each interbank reference to some symbol, another artificial symbol is introduced (e.g. by adding a prefix). This symbol will be defined by the linker and is located in a nonbanked area (which can be reached from all banks). It stands for a trampoline call where the current bank is stored, the bank is switched to the target bank and finally the jump to the target address is done.

For all that, the linker generates a temporary object file containing the definitions of the surrogate symbols and the program-specific trampoline code. The glue code for bank switching comes from a library which is searched after inserting the generated banking object file.

Of course, the specifics (and especially the code!) for banking heavily depend on the target platform. Hence the generic routines in the banking module rely on a variable which describes all characteristics of the target platform needed for banking. This variable tells the names of the generic banked area and the nonbanked area for the trampoline calls, how to construct a real banked area name from the bank number and most importantly a routine which generates the trampoline code for a single call. Note that the latter is somewhat tricky because this routine must generate object code (which references indexes of symbols...).

The variable is set when the target platform plugin is initialized. When banking is not used, the variable is null.

include "globdefs.h"
include "string.h"
include "stringlist.h"

typedef int Target_Bank;

type defined here to break circular include when importing "target.h"

typedef void (∗Banking_CallTemplateProc) (**in** UINT16 startAddress,
in UINT16 referencedAreaIndex,
in UINT16 targetSymbolIndex,
in UINT16 jumpLabelSymbolIndex,
out String_Type ∗codeSequence);

routine type for constructing the code for a trampoline call in the nonbanked code area; the start address of the code within the segment is given as `startAddress`*, the target symbol is given by* `targetSymbolIndex` *in area with index* `referencedAreaIndex` *and the jump label symbol as index* `jumpLabelSymbolIndex` *in the same area; the routine returns several code lines in* `codeSequence`*: a T line with the call and a relocation line referencing the target symbol and the bank switch label*

e.g. in the GBZ80 implementation for a call to routine XYZ in bank 23 the following trampoline call code (in assembler notation) is generated

BC_XYZ: LD HL,#XYZ // JMP Banking_switchTo_23

this means that a definition for "BC_XYZ" in the nonbanked area is used and a reference to "Banking_switchTo_23" in banked area 23; the code sequence effectively consists of six bytes (a 16 bit load and an absolute jump) represented by a pair of a T and R line

typedef void (∗Banking_NameConstructionProc) (**out** String_Type ∗name,
in Target_Bank bank);

routine type for constructing `name` *from* `bank` *(e.g. for a jump label for some bank)*

typedef void (∗Banking_SurrogateNameProc) (
out String_Type ∗surrogateSymbolName,
in String_Type symbolName);

routine type for constructing the surrogate symbol name `surrogateSymbolName` *used for a trampoline call from* `symbolName`

typedef Boolean (∗Banking_TargetValidationProc) (
in String_Type moduleName,
in String_Type segmentName,
in String_Type symbolName);

routine type for checking that some symbol is a valid target for an interbank call (i.e. it has to lie in a code segment) where symbol is characterized by `symbolName`*,* `segmentName` *and* `moduleName`

typedef struct {
String_Type genericBankedCodeAreaName;
/∗∗ name of generic area used for banked code symbols when bank assignment is not yet done (e.g. "CODE_0") ∗/
String_Type nonbankedCodeAreaName;
/∗∗ name of area used for nonbanked code symbols (e.g. "BASE") ∗/
Banking_NameConstructionProc makeBankedCodeAreaName;
/∗∗ template routine for constructing a banked code area name from

```
        a bank (also handling undefined bank correctly) */
  Banking_NameConstructionProc makeJumpLabelName;
    /** template routine for constructing the jump label (for bank
        switching within a trampoline call) from a bank */
  Banking_CallTemplateProc makeTrampolineCallCode;
    /** template routine for constructing a concrete trampoline call
        code */
  Banking_SurrogateNameProc makeSurrogateSymbolName;
    /** template routine for constructing a concrete trampoline
        surrogate symbol name from a symbol name */
  Banking_TargetValidationProc ensureAsCallTarget;
    /** template routine for checking that some symbol is a valid
        target for an interbank call (i.e. it has to lie in a code
        segment) */
  UINT8 offsetPerTrampolineCall;
    /** number of code bytes used for the trampoline call (which is
        fixed, because no code relaxation will occur) */
} Banking_Configuration;
```

void Banking_initialize (**void**);

sets up internal data structures for this module

void Banking_finalize (**void**);

cleans up internal data structures for this module

void Banking_adaptAreaNameWhenBanked (**in** /*Module_Type*/ **void** *module,
inout String_Type *areaName);

returns adapted `areaName` *whenever multiple conditions hold: banking is active, the area name specifies the generic banked area and additionally* `module` *is banked*

Boolean Banking_isActive (**void**);

tells whether banking is used at all

Target_Bank Banking_getModuleBank (**in** String_Type moduleName);

returns associated bank for module given by `moduleName` *or* `undefinedBank` *if none exists*

void Banking_readConfigurationFile (**in** String_Type fileName);

reads assignments of module to bank from file given by `fileName`*; the file consists of lines of assignments "modulename = bank"*

Boolean Banking_resolveInterbankReferences (**inout** StringList_Type *fileList);

traverses symbol list for interbank references; if such are found, a temporary object file is generated containing the trampoline code and its name is added to `fileList`*; returns false when no interbank reference has occured*

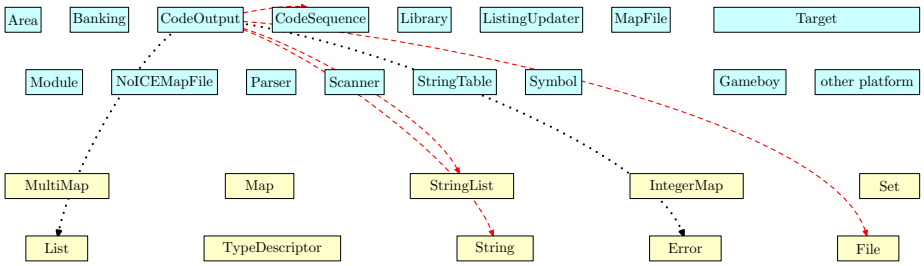

Figure 6: Module Dependencies of "CodeOutput" Module

6.3.3 Module "CodeOutput"

This module provides a generic service for putting out code sequences to file in the generic SDCC linker and some standard implementations for this service (like putting out Intel Hex format).

There are several code output streams available which have to be activated and get some code output routine assigned. Other modules feed code sequences into the CodeOutput module and it dispatches them to all listening code output streams.

For convenience the code output routines for Intel Hex and Motorola S19 format are provided. When those formats are not okay for some platform an own routine must be provided in the platform specific module and registered upon startup. An example can be found for the Gameboy platform which uses some simple binary memory dump format.

based on the module lkihx.c by Alan R. Baldwin

include "globdefs.h"
include "codesequence.h"
include "file.h"
include "string.h"
include "stringlist.h"

typedef enum {
  CodeOutput_State_atBegin, CodeOutput_State_inCode, CodeOutput_State_atEnd
} CodeOutput_State;

state where an output proc may be called: `atBegin` *is for any processing before output of the first code sequence,* `inCode` *is when putting out some intermediate code line and* `atEnd` *is for putting out the final record*

typedef void (∗CodeOutput_Proc)(**inout** File_Type ∗file,
in CodeOutput_State state,
in Boolean isBigEndian,
in CodeSequence_Type sequence);

type representing a routine to put out a code sequence processed by linker; `file` *is the file descriptor of the executable file,* `state` *tells whether the processing*

is started, in code processing or done, `isBigEndian` *tells the endianness of the target platform and* `sequence` *is the code sequence to be put out (when state is* `inCode`*)*

void CodeOutput_initialize (**in** Boolean targetIsBigEndian);

initializes internal data structures; `targetIsBigEndian` *tells about the endianness of the target platform*

void CodeOutput_finalize (**void**);

cleans up internal data structures

Boolean CodeOutput_create (**in** String_Type fileName,
in CodeOutput_Proc outputProc);

creates another code output stream on file with `filename` *with a routine formatting the code sequences* `outputProc`*; when opening the file for writing fails, the routine returns false*

void CodeOutput_closeStreams (**void**);

puts the terminating record to all open code output streams and additionally closes all code stream files

void CodeOutput_getFileNames (**out** StringList_Type *fileNameList);

returns list of file names for all registered output streams

void CodeOutput_writeLine (**in** CodeSequence_Type sequence);

puts the representation of code sequence `sequence` *to all open code output streams*

void CodeOutput_writeIHXLine (**inout** File_Type *file, **in** CodeOutput_State state,
in Boolean isBigEndian,
in CodeSequence_Type sequence);

predefined code output routine producing Intel Hex format

void CodeOutput_writeS19Line (**inout** File_Type *file, **in** CodeOutput_State state,
in Boolean isBigEndian,
in CodeSequence_Type sequence);

predefined code output routine producing Motorola S19 format

6.3.4 Module "CodeSequence"

This module provides all services for scanning code and relocating it in the generic SDCC linker.

As the name indicates it encapsulates code sequences, i.e. byte sequences to be put to some bank at some specific address.

Often relocation is needed. This is modeled as a sequence of simple relocations, where each simple relocation specifies some position in a code sequence, an offset value and some indication on how the value pointed to has to be combined with the offset. The central routine of this module applies a relocation list to some code sequence.

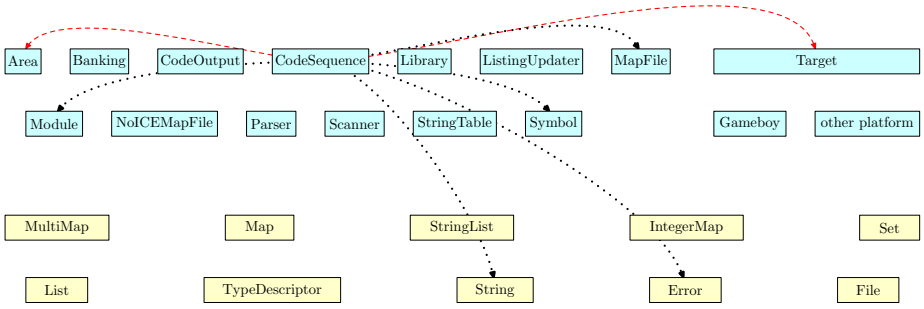

Figure 7: Module Dependencies of "CodeSequence" Module

based on the module lkrloc.c by Alan R. Baldwin

include "globdefs.h"
include "target.h"

define CodeSequence_maxLength ...
maximum length of single codesequence to be relocated

typedef struct {
/ Area_Segment segment; //changed to break circular includes */*
struct Area__SegmentRecord *segment;
Target_Bank romBank;
UINT32 offsetAddress;
UINT8 length; */** *number of bytes in sequence */*
UINT8 byteList[CodeSequence_maxLength];
} CodeSequence_Type;
type representing a code sequence before or after relocation

typedef struct {
Boolean msbByteIsUsed : 1;
Boolean isThreeByteAddress : 1;
Boolean mostSignificantByteIsUsed : 1;
Boolean pageIsReferenced : 1;
Boolean zeroPageIsReferenced : 1;
Boolean dataIsSigned : 1;
Boolean slotWidthIsTwo : 1;
Boolean isRelocatedPCRelative : 1;
Boolean isSymbol : 1;
Boolean elementsAreBytes : 1;
} CodeSequence_RelocationKind;
relocation attribute for a single relocation entry

typedef struct {
CodeSequence_RelocationKind kind;
UINT8 index;

```
  UINT16 value;
} CodeSequence_Relocation;
```

a single relocation for a previous code line has a relocation `kind`*, an* `index` *into the unrelocated code line and a relocation* `value` *(typically some offset)*

```
typedef struct {
  /* Area_Segment segment; //changed to break circular includes */
  struct Area__SegmentRecord *segment;
  CodeSequence_Relocation list[CodeSequence_maxLength];
  UINT8 count;
} CodeSequence_RelocationList;
```

a relocation list for a code line has a `segment` *of relocation and several relocations and a count of relocations*

void CodeSequence_initialize (**void**);

sets up internal data structures

void CodeSequence_finalize (**void**);

cleans up internal data structures

void CodeSequence_makeKindFromInteger (**out** CodeSequence_RelocationKind *kind,
in UINT8 value);

makes a relocation kind `kind` *from some external representation* `value`

void CodeSequence_relocate (**inout** CodeSequence_Type *sequence,
in UINT16 areaMode,
in CodeSequence_RelocationList *relocationList);

relocates code line `sequence` *based on information in an R line of the linker input; an R line contains an area index specifying the associated area, a mode for that area and a list of four byte relocation data: that information is given in* `areaMode` *and* `relocationList`

UINT8 CodeSequence_convertToInteger (**in** CodeSequence_RelocationKind kind);

converts relocation kind `kind` *to external integer representation*

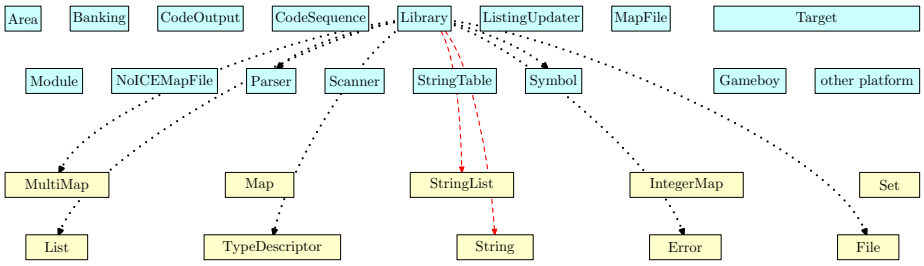

Figure 8: Module Dependencies of “Library” Module

6.3.5 Module "Library"

This module provides all services for object file libraries in the generic SDCC linker.

Object file libraries are searched for in directories given by the (inbound) routine `addDirectory`*. The names of those libraries are also specified by the routine* `addFilePathName`*.*

A call to the central routine `resolveUndefinedSymbols` *searches all matching files for symbol definitions. Those definitions are used to satisfy unresolved references from the object modules linked so far. This process is repeated until no more resolutions can be done. All encountered symbols are added to the linker symbol table and the associated code from the libraries can be added to the code later.*

Note that SDCCLIBs with an XML-structure are not yet supported.

based on the module lklibr.c by Alan R. Baldwin

include "globdefs.h"
include "string.h"
include "stringlist.h"

void Library_initialize (**void**);
initializes the internal data structures of the library manager

void Library_finalize (**void**);
cleans up the internal data structures of the library manager

void Library_getFileNameList (**out** StringList_Type *libraryFileNameList);
returns list of library object files used so far in `libraryFileNameList`

void Library_addCodeSequences (**void**);
adds code defined in all referenced library object files

void Library_addDirectory (**in** String_Type path);
adds some directory `path` *to the list of paths*

void Library_addFilePathName (**in** String_Type path, **out** Boolean *isFound);
adds some library file relative or absolute `path`*; returns in* `isFound` *whether library has been found in directory path list or not*

void Library_resolveUndefinedSymbols (**void**);
searches all specified library files and library directories for undefined symbols until no more resolutions can be done; adds newly referenced symbols to symbol table and keeps track of all used library files

6.3.6 Module "ListingUpdater"

This module provides all services for augmenting listing files.

Based on the list of link files, the central routine `update` *scans the appropriate directories for associated assembler listings and inserts the relocated code at appropriate*

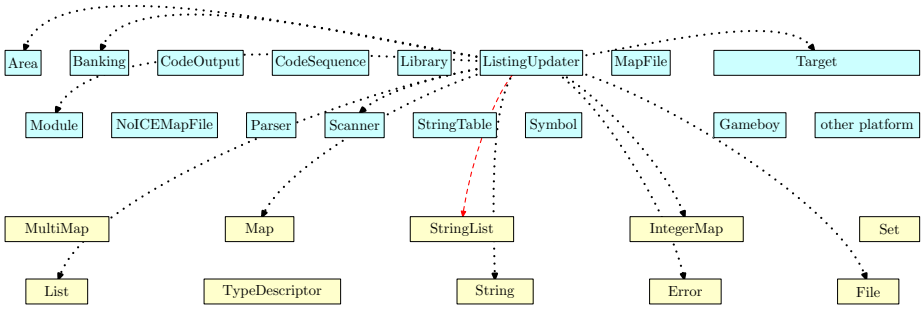

Figure 9: Module Dependencies of "ListingUpdater" Module

places. The code is queried from the Target module and is only available at the end of the second linking pass.

Note that the module knows very much about the structure of a assembler listing file and is fragile whenever that structure changes in the future.

include "globdefs.h"
include "stringlist.h"

void ListingUpdater_initialize (**void**);
sets up internal data structures for this module

void ListingUpdater_finalize (**void**);
cleans up internal data structures for this module

void ListingUpdater_update (**in** UINT8 base, **in** StringList_Type linkFileList);
update listings with radix `base` *and list of linked files* `linkFileList`

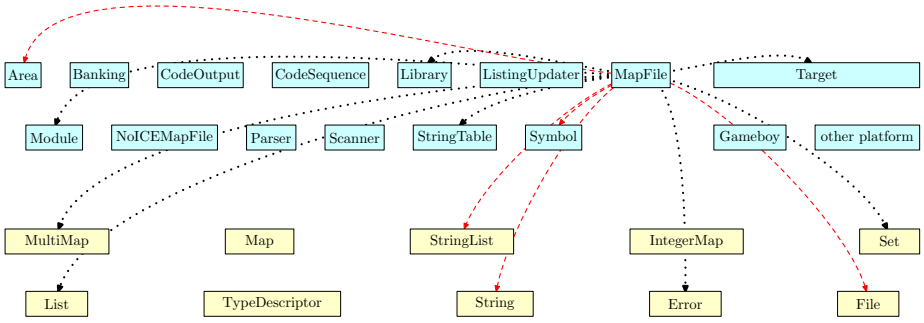

Figure 10: Module Dependencies of "MapFile" Module

6.3.7 Module "MapFile"

This module provides all services for putting out mapfiles. Those map files give an overview about the object files read, the allocation of symbols and areas and the library files used.

Several map files can be open at once and they may have different output routines. The map target files and the routines have to be registered in this module and they are automatically activated when all map information is available. The standard map file is produced via the output routine `generateStandardFile`*, but there are also variants possible. One can be found in the Gameboy module, where a map file for the NoGMB emulator is produced.*

```
include "area.h"
include "file.h"
include "globdefs.h"
include "string.h"
```

```
typedef void (*MapFile_CommentOutputProc)(inout File_Type *file,
                                          in String_Type comment);
```

type representing a routine to conditionally add text from magic comments to some map file

```
typedef void (*MapFile_SymbolTableOutputProc)(inout File_Type *file);
```

type representing a routine to produce a map file of the linker output; `file` *is the file descriptor of the map file which is already open and will be closed outside that routine*

```
typedef struct {
  MapFile_CommentOutputProc commentOutputProc;
  MapFile_SymbolTableOutputProc symbolTableOutputProc;
} MapFile_ProcDescriptor;
```

```
void MapFile_initialize (void);
```

sets up internal data structures for this module

```
void MapFile_finalize (void);
```

cleans up internal data structures for this module

```
Boolean MapFile_isOpen (void);
```

tells whether map files are open or not

```
void MapFile_getSortedAreaSymbolList (in Area_Type area,
                                      out Symbol_List *areaSymbolList);
```

collects all symbols in `area` *and returns them in* `areaSymbolList` *sorted by address; this is merely a convenience routine to be used by mapfile generators*

```
void MapFile_registerForOutput (in String_Type fileNameSuffix,
                                in MapFile_ProcDescriptor routines);
```

registers `routines` *for mapfile output to file with map file specific extension* `fileNameSuffix`

void MapFile_openAll (**in** String_Type fileNamePrefix);
opens all map files with names given by `fileNamePrefix` *plus a map file specific extension*

void MapFile_closeAll (**void**);
closes all open map files

void MapFile_setOptions (**in** UINT8 base, **in** StringList_Type linkFileList);
sets options for map file output to radix `base` *and list of linked files* `linkFileList`

void MapFile_writeErrorMessage (**in** String_Type message);
writes `message` *as warning to all currently open map files*

void MapFile_writeSpecialComment (**in** String_Type comment);
writes `comment` *to all currently open map files if relevant for them*

void MapFile_writeLinkingData (**void**);
writes linker symbol information to all currently open map files

void MapFile_generateStandardFile (**inout** File_Type *file);
generates canonical map file output into `file`

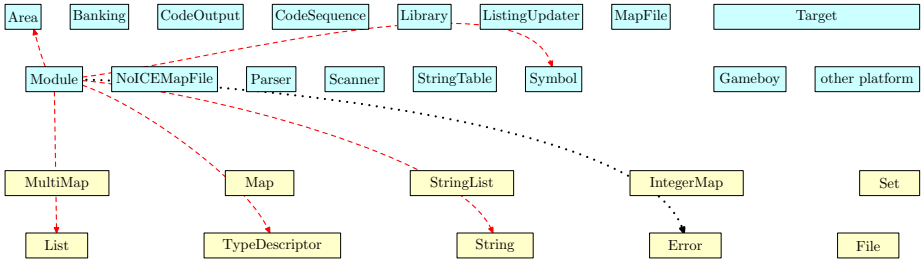

Figure 11: Module Dependencies of "Module" Module

6.3.8 Module "Module"

This module provides all services for module definitions in the generic SDCC linker.

A module is a group of code and data areas belonging together and is the root of all related linker objects. Normally a module is defined by a single object file, but in case of a library several object files may contribute to a single module.

Modules may have associated areas and symbols and appropriate routines provide access to those lists.

There is also the notion of a current module, which is the one where the currently parsed symbols and areas are associated to.

based on the module lkhead.c by Alan R. Baldwin

typedef struct Module__Record *Module_Type;
type representing a module (as an opaque type)

include "area.h"
include "file.h"
include "globdefs.h"
include "list.h"
include "string.h"
include "symbol.h"

typedef UINT16 Module_SegmentIndex;
type for unique numbers of segments per module

typedef UINT16 Module_SymbolIndex;
type for unique numbers of symbols per module

extern TypeDescriptor_Type Module_typeDescriptor;
variable used for describing the type properties when module objects occur in generic types like lists

void Module_initialize (**void**);
sets up all internal data structures

void Module_finalize (**void**);
cleans up all internal data structures

void Module_make (**in** String_Type associatedFileName,
in Module_SegmentIndex segmentCount,
in Module_SymbolIndex symbolCount);
creates a new module structure and links it into the list of module structures; `associatedFileName` *tells the file name,* `segmentCount` *and* `symbolCount` *tell how many area segments and symbols are in this module*

void Module_destroy (**inout** Module_Type *module);
deallocates `module`

void Module_getFileName (**in** Module_Type module, **out** String_Type *fileName);
returns associated file name of `module` *in* `fileName`

void Module_getName (**in** Module_Type module, **out** String_Type *name);
returns name of `module` *in* `name`

Area_Segment Module_getSegment (**in** Module_Type module,
in Module_SegmentIndex segmentIndex);
returns segment with index `segmentIndex` *within* `module` *or NULL if not found*

Area_Segment Module_getSegmentByName (**in** Module_Type module,
in String_Type segmentName);
returns segment with name `segmentName` *within* `module` *or NULL if not found*

Symbol_Type Module_getSymbol (**in** Module_Type module,
in Module_SymbolIndex symbolIndex);
returns symbol with index `symbolIndex` *within* `module` *or NULL if not found*

Symbol_Type Module_getSymbolByName (**in** Module_Type module,
in String_Type symbolName);
returns symbol with name `symbolName` *within* `module` *or NULL if not found*

Module_Type Module_currentModule (**void**);
returns currently active module

void Module_getModuleList (**inout** List_Type ∗moduleList);
returns the list of all modules in `moduleList`

void Module_getSegmentList (**in** Module_Type module,
inout Area_SegmentList ∗segmentList);
returns the list of all segments within `module` *in* `segmentList`

void Module_getSymbolList (**in** Module_Type module,
inout Symbol_List ∗symbolList);
returns the list of all symbols within `module` *in* `symbolList`

void Module_setCurrentByName (**in** String_Type name,
out Boolean ∗isFound);
select current module by associated module name `name`*;* `isFound` *tells whether search has been successful*

void Module_setCurrentByFileName (**in** String_Type fileName,
out Boolean ∗isFound);
select current module by associated file name `fileName`*;* `isFound` *tells whether search has been successful*

void Module_setName (**in** String_Type name);
sets the name of the current module to `name`

void Module_addSegment (**inout** Module_Type ∗module, **in** Area_Segment segment);
adds `segment` *to* `module` *and returns that module*

void Module_addSymbol (**inout** Module_Type ∗module, **in** Symbol_Type symbol);
adds `symbol` *to* `module`

void Module_replaceSymbol (**inout** Module_Type ∗module, **in** Symbol_Type oldSymbol,
in Symbol_Type newSymbol);
replaces `oldSymbol` *in symbol list of* `module` *by* `newSymbol`*; does nothing when* `oldSymbol` *does not occur*

void Module_toString (**in** Module_Type module, **out** String_Type ∗representation);
constructs a printable representation of `module`*, its internal data and its associated segments (for debugging purposes) and concatenates it to* `representation`

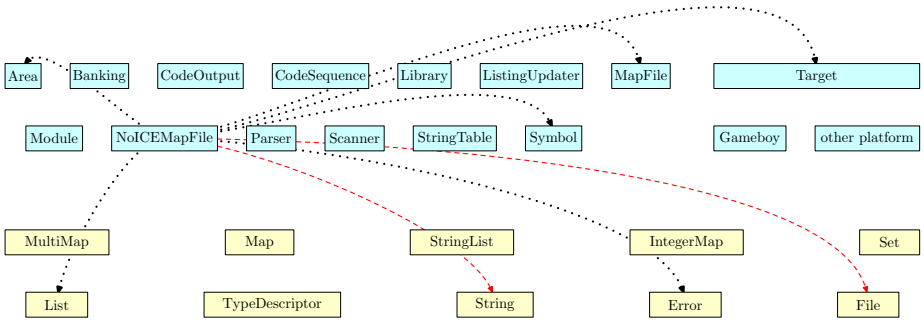

Figure 12: Module Dependencies of "NoICEMapFile" Module

6.3.9 Module "NoICEMapFile"

This module provides all services for putting out mapfiles in NoICE format. The output routine defined here links into the generic MapFile module of the SDCC linker.

based on module lknoice.c by John Hartman

include "file.h"
include "globdefs.h"

void NoICEMapFile_initialize (**void**);
sets up internal data structures for this module

void NoICEMapFile_finalize (**void**);
cleans up internal data structures for this module

void NoICEMapFile_addSpecialComment (**inout** File_Type *file,
in String_Type comment);
adds `comment` *conditionally to NOICE* `file` *when it contains a relevant information*

void NoICEMapFile_generate (**inout** File_Type *file);
writes a map file in NoICE format to `file`

6.3.10 Module "Parser"

This module provides all services for parsing tokenized character streams in the generic SDCC linker.

The parser can parse a single object file or a list of them. It normally calls other modules to build up internal object networks for an object file, but it also can on request do a reduced scan and simply return a list of symbols encountered e.g. when reading a library file.

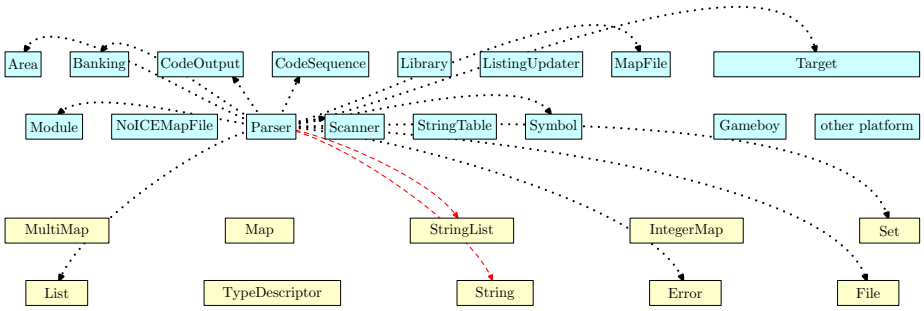

Figure 13: Module Dependencies of "Parser" Module

Other services can parse value assignment equations between some identifier and a long integer value. Those parsing routines are generic because they do a callback to some key-value-assignment routine. Note that in contrast to the original linker those routines cannot parse integer RHS expressions, but only simple values.

include "globdefs.h"
include "stringlist.h"

typedef struct {
  UINT8 defaultBase; */* the default base of number strings read */*
  enum { littleEndian, bigEndian, unknown} endianness;

typedef void (*Parser_KeyValueMappingProc)(**in** String_Type key,
in long value);

callback routine type for mapping string `key` *to integer* `value` *to be used in* `scanStringList`

void Parser_initialize (**void**);

initializes the internal data structures of the parser

void Parser_finalize (**void**);

cleans up the internal data structures of the parser

void Parser_collectSymbolDefinitions (**in** String_Type objectFileName,
inout StringList_Type *symbolNameList);

parses file given by `objectFileName` *for symbol definitions in command S and returns them in* `symbolNameList`

void Parser_setDefaultOptions (**in** Parser_Options options);

sets the options for subsequent parsing

void Parser_parseObjectFile (**in** Boolean isFirstPass, **in** String_Type fileName);

parses the object file given by `fileName` *for commands X, D, Q, H, M, A, S, T, R, and P; depending on whether this is in the first or second pass the processing is different*

void Parser_parseObjectFiles (**in** Boolean isFirstPass,
in StringList_Type fileNameList);

parses the object files in `fileNameList` *for commands X, D, Q, H, M, A, S, T, R, and P; depending on whether this is in the first or second pass the processing is different*

void Parser_setMappingFromList (**in** StringList_Type valueMapList,
in Parser_KeyValueMappingProc setElementValueProc);

parses the string list in `valueMapList` *for lines of the form "name=value" and calls* `setElementValueProc` *for each (name,value) pair*

void Parser_setMappingFromString (**in** String_Type valueMapString,
in Parser_KeyValueMappingProc setElementValueProc);

parses the string in `valueMapString` *as a lines of the form "name=value" and calls* `setElementValueProc` *for given (name,value) pair*

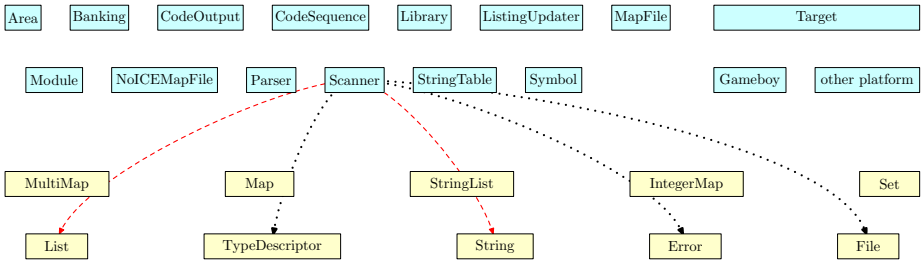

Figure 14: Module Dependencies of "Scanner" Module

6.3.11 Module "Scanner"

This module provides all services for tokenizing character streams in the generic SDCC linker.

Tokenization is done on some input stream specified via a reader routine. A token consists of a kind, an operator (when kind is "operator") and some external string representation.

Tokens may be pushed back when doing a lookahead during parsing.

include "globdefs.h"
include "list.h"
include "string.h"
include "typedescriptor.h"

define Scanner_endOfStreamChar ...

character for telling that the end of the input stream has been reached

define Scanner_pushbackStackSize ...

maximum number of tokens pushed back for rereading

typedef char (∗Scanner_ReaderProc)(**void**);

callback routine for reading next character on some input stream; returns `endOfStreamChar` *when end of stream is reached*

typedef List_Type Scanner_TokenList;

list of tokens

```
typedef enum {
  Scanner_TokenKind_operator, Scanner_TokenKind_identifier,
  Scanner_TokenKind_number, Scanner_TokenKind_idOrNumber,
  Scanner_TokenKind_newline, Scanner_TokenKind_streamEnd,
  Scanner_TokenKind_comment, Scanner_TokenKind_other
} Scanner_TokenKind;
```

kinds of tokens known by the scanner; `idOrNumber` *is an ambiguous symbol which consists of hexadecimal characters only*

```
typedef enum {
  Scanner_Operator_plus, Scanner_Operator_minus, Scanner_Operator_times,
  Scanner_Operator_div, Scanner_Operator_mod, Scanner_Operator_shiftLeft,
  Scanner_Operator_shiftRight, Scanner_Operator_or, Scanner_Operator_and,
  Scanner_Operator_complement, Scanner_Operator_assignment,
  Scanner_Operator_other
} Scanner_Operator;
```

different operator tokens

```
typedef struct {
  Scanner_TokenKind kind;
  String_Type representation;
  Scanner_Operator operator;
} Scanner_Token;
```

token returned by the scanner

void Scanner_initialize (**void**);

initializes the internal data structures of the scanner

void Scanner_finalize (**void**);

cleans up the internal data structures of the scanner

void Scanner_makeToken (**out** Scanner_Token ∗token);

initializes `token`

void Scanner_makeTokenList (**out** Scanner_TokenList ∗tokenList,
in String_Type st);

scans `st` *and returns all tokens found in* `tokenList`

void Scanner_destroyToken (**inout** Scanner_Token ∗token);

finalizes `token`

void Scanner_getNextToken (**inout** Scanner_Token ∗token);

returns next token on current input stream in `token`

void Scanner_ungetToken (**in** Scanner_Token token);

pushes back `token` *to current input stream; this may be repeatedly called up to a limit of* `pushbackStackSize` *tokens simultaneously pushed back*

void Scanner_tokenToString (**in** Scanner_Token token, **out** String_Type *st);

returns representation of `token` *in* `st`

void Scanner_redirectInput (**in** Scanner_ReaderProc readerProc);

tells that `readerProc` *is the new routine for getting at the next character*

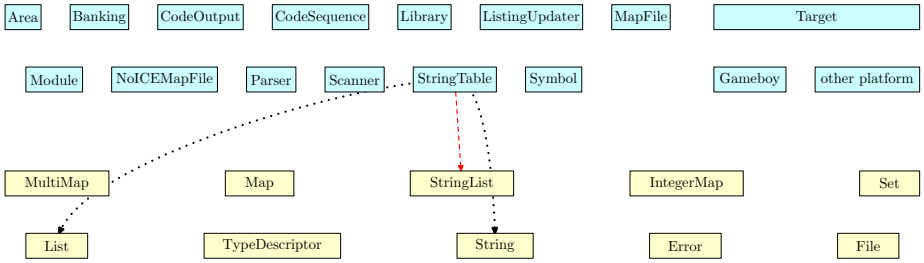

Figure 15: Module Dependencies of "StringTable" Module

6.3.12 Module "StringTable"

This module provides all services for handling two string tables in the SDCC linker. Those string tables contain the global base address definitions and the global symbol definitions as strings.

include "globdefs.h"
include "stringlist.h"

typedef StringList_Type StringTable_Type;

extern StringTable_Type StringTable_baseAddressList;

extern StringTable_Type StringTable_globalDefList;

void StringTable_initialize (**void**);

sets up internal data structures for this module

void StringTable_finalize (**void**);

cleans up internal data structures for this module

void StringTable_addCharArray (**inout** StringTable_Type *stringTable,
in char *st);

adds character array `st` *to string table* `stringTable`

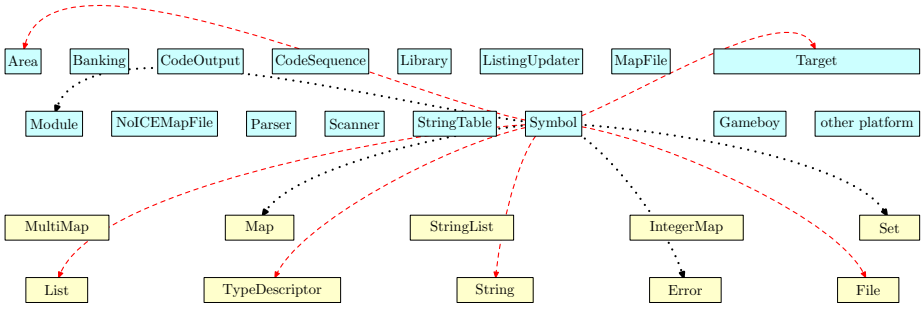

Figure 16: Module Dependencies of "Symbol" Module

6.3.13 Module "Symbol"

This module provides all services for handling external symbols within the SDCC linker.

A symbol has a name, some segment and address which may be set and queried. When a symbol is newly introduced one has to tell whether this introduction is a symbol definition or symbol reference. This property may also be queried.

For interbank calls a symbol may be split into a real and a surrogate symbol. It is also possible to get a list of referenced but undefined symbols (necessary when checking the libraries).

include "file.h"
include "globdefs.h"
include "list.h"
include "set.h"
include "string.h"
include "target.h"
include "typedescriptor.h"

typedef struct Symbol_Record *Symbol_Type;

a external or internal symbol within the linker (as an opaque type)

typedef List_Type Symbol_List;

a list of symbols

include "area.h"

extern TypeDescriptor_Type Symbol_typeDescriptor;

variable used for describing the type properties when symbol objects occur in generic types like lists

void Symbol_initialize (**in** Boolean platformIsCaseSensitive);

sets up all internal data structures; `platformIsCaseSensitive` *tells whether case matters for identifiers or not*

void Symbol_finalize (**void**);
cleans up all internal data structures

Symbol_Type Symbol_make (**in** String_Type symbolName, **in** Boolean isDefinition,
in Target_Address startAddress);
makes a new symbol with `symbolName`*; additionally it is specified whether this is a definition and the* `startAddress`

Symbol_Type Symbol_makeBySplit (**in** Symbol_Type oldSymbol,
in String_Type symbolName);
splits `oldSymbol` *and creates new symbol with* `symbolName`*; all references to* `oldSymbol` *are transferred to new symbol; the new symbol is referenced but undefined, while the old symbol is defined but unreferenced*

void Symbol_destroy (**inout** Symbol_Type ∗symbol);
destroys `symbol`

void Symbol_getName (**in** Symbol_Type symbol, **out** String_Type ∗name);
returns name of `symbol` *in* `name`

Area_Segment Symbol_getSegment (**in** Symbol_Type symbol);
returns segment of `symbol`

Boolean Symbol_isDefined (**in** Symbol_Type symbol);
tells whether `symbol` *is defined in some module*

Boolean Symbol_isSurrogate (**in** Symbol_Type symbol);
tells whether `symbol` *is a surrogate symbol (used for banking)*

void Symbol_setAddressForName (**in** String_Type symbolName,
in Target_Address address);
sets address of existing symbol with `symbolName` *to* `address`

Symbol_Type Symbol_lookup (**in** String_Type symbolName);
returns `symbol` *with* `symbolName` *or NULL when not found*

Target_Address Symbol_absoluteAddress (**in** Symbol_Type symbol);
returns absolute address of `symbol` *(by adding the segment base address)*

void Symbol_getUndefinedSymbolList (**inout** Symbol_List ∗undefinedSymbolList);
returns list of undefined symbols in `undefinedSymbolList`

void Symbol_checkForUndefinedSymbols (**inout** File_Type ∗file);
scans the table of symbols for referenced but undefined symbols; for each of those symbols a message is output to `file` *telling the module where a reference has been made*

void Symbol_toString (**in** Symbol_Type symbol, **out** String_Type ∗representation);
constructs a printable representation of `symbol` *and its internal data (for debugging purposes) and concatenates it to* `representation`

6.4 Platform Modules

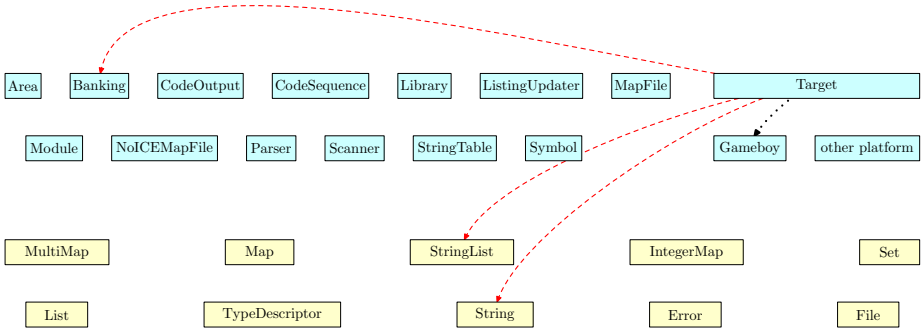

Figure 17: Module Dependencies of "Target" Module

6.4.1 Module "Target"

This module provides all services for specifying target specific configuration information within the SDCC linker.

include "banking.h"
include "globdefs.h"
include "string.h"
include "stringlist.h"

typedef UINT16 Target_Address;
target addresses are 16 bit

define Target_undefinedBank ...
undefined value for a bank number

typedef int Target_Bank;
bank number type

typedef Target_Bank (*Target_BankAnalysisProc) (**in** String_Type segmentName);
type for routines parsing the current segment with `segmentName` *of emitted code for ROM bank switching*

typedef UINT8 (*Target_CodeQueryProc)(**in** Target_Bank bank,
in Target_Address address);
type for routines returning the associated emitted code byte for `bank` *and* `address`

typedef void (*Target_CommandLineHandleProc)(**in** String_Type mainFileNamePrefix,
in StringList_Type argumentList,
inout Boolean optionIsHandledList[]);

type for routines parsing the command line options in `argumentList` *for options relevant for this platform; all options where* `optionIsHandledList` *is true have already been processed before; when an option is used by that routine,* `optionIsHandledList` *is also set to true;* `mainFileNamePrefix` *tells the name of the main file passed as parameter without extension*

typedef void (∗Target_UsageInfoProc)(**out** String_Type ∗st);

type for routines returning a string with an indented line list (separated by newlines) with platform specific options as a usage info

typedef void (∗Target_InitializationProc)(**void**);

type for routines setting up module internal data

typedef void (∗Target_FinalizationProc)(**void**);

type for routines cleaning up module internal data

typedef struct {
  Boolean isBigEndian;
  Boolean isCaseSensitive;
  Target_BankAnalysisProc getBankFromSegmentName;
  Target_CodeQueryProc getCodeByte;
  Target_UsageInfoProc giveUsageInfo;
  Target_CommandLineHandleProc handleCommandLineOptions;
  Target_InitializationProc initialize;
  Target_FinalizationProc finalize;
  Banking_Configuration ∗bankingConfiguration;
} Target_Type;

type to tell several properties of target platform like endianness, case sensitivity of names, banking configuration, callback routines for rom bank switching, querying for bytes in the emitted code, command line option parsing, giving usage information for target specific options and setting up and tearing down the platform specific data; each of those routines may be NULL when it is not used in this target platform

extern Target_Type Target_info;

variable containing the information about the current target platform

void Target_initialize (**void**);

sets up module internal data

void Target_finalize (**void**);

cleans up module internal data

void Target_setInfo (**in** String_Type platformName);

sets `info` *for platform specified by* `platformName`

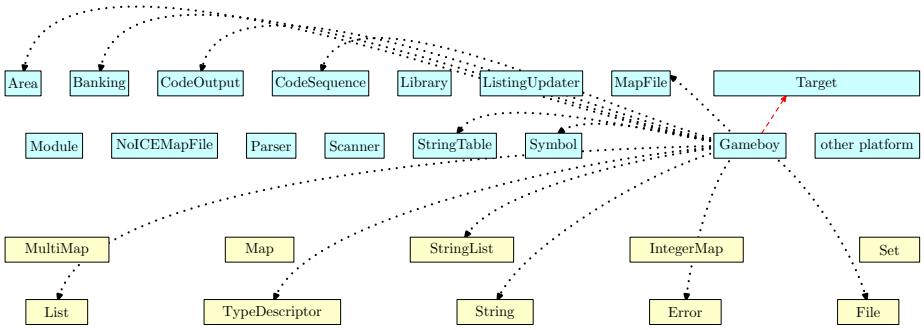

Figure 18: Module Dependencies of “Gameboy” Module

6.4.2 Module “Gameboy target”

Gameboy This module provides the target specific services for the Gameboy target within the generic SDCC linker.

based on the module lkgb.c by Pascal Felber

include ”../target.h”

extern Target_Type Gameboy_targetInfo;

7 Appendix: File Formats

7.1 Object File

The following EBNF-grammar gives the token structure of an object file used as input to the linker.

Tokens are either operators, identifiers, numbers, newlines and comments. White space (like blanks or tabulators) is necessary to seperate numbers and identifiers. Note that a newline is no valid whitespace but a token on its own.

address	::=	word .
areaCount	::=	number .
areaIndex	::=	word .
areaMode	::=	word .
areaDefinition	::=	areaLine { symbolLine } { codeLine relocLine } .
areaLine	::=	'A' areaName 'size' number 'flags' number newline .
codeLine	::=	'T' address { number } newline .
endiannessChar	::=	'H' | 'L' .
headerLine	::=	'H' areaCount 'areas' symbolCount 'global' 'symbols' newline .
moduleLine	::=	'M' moduleName newline .
moduleName	::=	identifier .
objectFile	::=	radixLine headerLine moduleLine [optionsLine] { areaDefinition } .
optionsLine	::=	'O' { (identifier | number | operator) } newline .
radixChar	::=	'X' | 'D' | 'Q' .
radixLine	::=	radixChar endiannessChar newline .
relocationData	::=	number .
relocationIndex	::=	number .
relocationInfo	::=	relocationKind relocationIndex relocationData .
relocationKind	::=	number .
relocLine	::=	'R' areaMode areaIndex { relocationInfo } newline.
symbolCount	::=	number .
symbolLine	::=	'S' identifier ('Def' | 'Ref') number newline .
word	::=	number number .

7.2 Library File

The following EBNF-grammar gives the token structure of an library file used as input to the linker.

Note that the revised linker does *not* support the SDCCLIB libraries containing directly embedded and indexed object files. The only files supported are libraries referencing external object files by name.

fileName	::=	≪string without newline character≫ newline .
libraryFile	::=	{ fileName newline } .

References

[Baldwin09] Alan Baldwin. *ASxxxx Cross Assemblers.* Kent State University, Kent, Ohio. (2009). http://shop-pdp.kent.edu/ashtml/asxxxx.htm