1

Development Chain

This chapter is focused on the software development chain for a microprocessor-based system. The different steps and their associated tools, including the first, the assembler, are examined. The debugging and testing aspects have taken on greater importance as hardware and software have grown more complex and become embedded in systems. These software and hardware tools are then described in detail. At first conceived to be used at the printed circuit level, they must be integrated progressively. To conclude, assembly language, a first level symbolic language, will be surveyed. The reader who is interested in a specific processor should refer to documentation from the relevant manufacturer. In addition, we will not here make a distinction between microprocessor and microcontroller because, in the latter case, we will only examine the "computation" component, that is, the processor, without addressing the other two subsystems, which are Input/Output (I/O) and RAM/ROM (Random Access/Read-Only Memory), including programmable memory, as relates to the latter type.

NOTE.– An understanding of the concepts of data representation and arithmetic operations in a computer is assumed. Otherwise, *cf*. Darche (2000). This will also be the case for logical operators. On this subject, *cf*. Darche (2002).

1.1. Layers of languages, stages of development and tools

The processor uses two-state logic. Instruction codes and data are therefore expressed in binary. This is machine language. Manipulating such data is not humanly possible for a program longer than about a hundred lines. The first computers were programmed in this language, and the program (i.e. machine code and data) was entered in binary format using switches as input peripherals. Because of the difficulty of use, an additional layer of language closer to natural language (English) was therefore necessary. This language is called "Assembly Language,"

which is abbreviated in this work as AL. The term assembly emerged from the EDSAC project (*Electronic Delay Storage Automatic Calculator*, Wilkes 1950, 1951) defining the action of reading subprograms from a symbolic instruction tape (a letter), translating them into binary, and making them executable by the main program (modern functions of a link editor and loading program combined). This mechanism is attributed to David J. Wheeler (Wheeler 1950). Excluding machine-oriented language, assembly language is therefore the processor's base programming language. It is referred to as symbolic. This indicates that it manipulates symbolic information, instruction words, variables and labels primarily. A specific assembly language corresponds to a single Instruction Set Architecture (ISA, *cf.* § V1-3.5). Assembly instructions are referred to using a form of abbreviated writing called operation code (or opcode) mnemonic (*cf.* § V4-2.1). An opcode mnemonic is a symbolic representation of a machine code. The assembler is the computing tool that translates between the symbolic name and the binary value. The first assembler was written by Nathaniel Rochester for the IBM 701 (1954). Recall that instructions in machine language, also called machine code or sometimes hard code (Bell 1973), are a series of binary instructions that are read and executed by the microprocessor. Figure 1.1 shows an example of assembly with two lines of instructions for the 8086 microprocessor. The assembler translated the mnemonics into machine code, here expressed in base 16 for readability. This tool is specialized for a processor or a family of processors. There is no efficiency loss between assembly languages and binary because the translation is direct. This is not the case for High-Level (programming) Language (HLL) compilers, where one high-level instruction (i.e. statement) corresponds to a sequence of several instructions in assembly language.

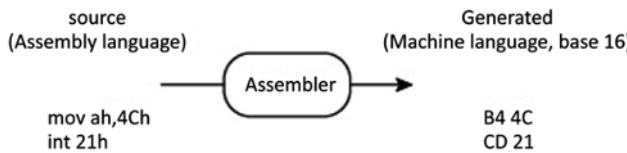

Figure 1.1. *Example of lines from a program written in 80x86 assembly language*

1.1.1. *Levels of languages*

Programming languages evolve in the direction of the increasing ease of programming by making it possible to manipulate higher-level abstractions, for example at the level of data structures or operators. Assembly language is downstream from high-level language, as shown in Figure 1.2. A language referred to as high level is as close as possible to human language. This is why machine and

assembly languages are called Low-Level (programming) Languages[1] (LLL). To move from one level to a lower level, it is necessary to use a translator that will substitute a source-language instruction with a series of instructions belonging to the lower-level (target) language. Each instruction in machine language that is executed will give rise to a series of commands inside the microprocessor, or micro-operations. In a micro-programmed architecture, these commands are naturally called micro-instructions (this will be covered in a future book by the author on microprocessors). This microprogramming language belongs to the language level reserved for processor designers (not represented).

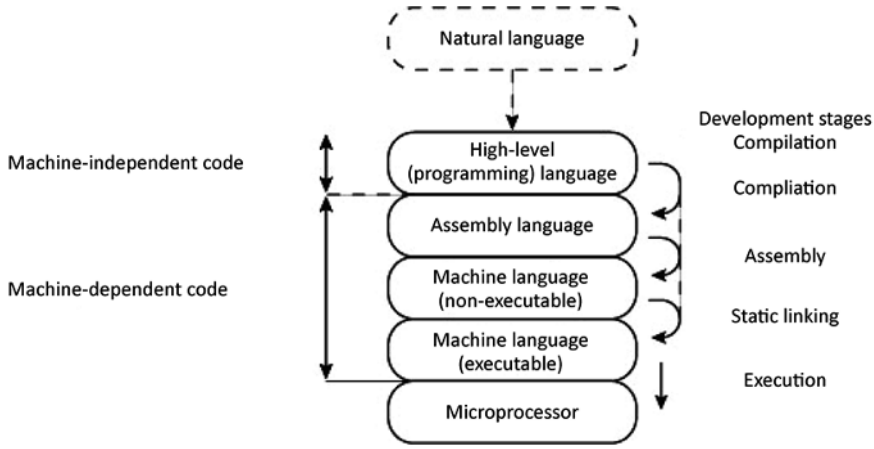

Figure 1.2. *Levels of languages in computing*

It should be noted that there are microcontrollers with on-board high-level language interpreters (*cf.* § V3-5.3).

1 High-level languages like C (Kernighan 1983) are sometimes classified by some authors (*cf.* Doyle (1985), for example) in the "low-level" category because they offer instructions close to the microprocessor such as sequential and combinatorial logic operators, and because they make it possible to manipulate variable addresses (concept of the pointer). This argument will not be adopted in this work because these languages provide high-level control structures and the idea of data typing, which is not the case for AL.

1.1.2. *Development stages*

Figure 1.3 presents the development chain for a computing application written in a compilable high-level language. It is also called the compilation chain or, more generally, the toolchain. A source program (or code) is written in a high-level language, for example in C (file extension .c) using a text editor. It is first precompiled (file extension .i for our example in the Microsoft environment or direct display on the standard output peripheral for UNIX). The precompiler is a preprocessor that will transform the source before delivering it to the compiler. It primarily provides functions for macro-expansion, macro(-definition) and file inclusion (*cf.* § 1.3.4). It also uses control structures to enable conditional compilation. Macro-operations are customizable thanks to its formal parameters, as are the functions, and they can be interlocked. The preprocessor is therefore no more or less than a manipulator of character strings with functions for search, deletion, insertion and substitution of character strings, just like a simple text editor. An additional function is the deletion of comments, which are useless to the machine. The transformed source is then compiled to obtain a file in assembly language (generated file extension .s or .asm, respectively under UNIX and Windows), which is then assembled. In simple cases, the obtained file is a file (example of an output file extension .hex), a binary memory image ready to be loaded into RAM or ROM (FW for FirmWare). The case of separate compilation (modular programming), or where the execution environment for the software is taken into account as, for example, with an Operating System (OS), generates a binary object file (output file extension .o or .obj depending on the OS), also referred to as an object module. An object program is therefore the final result from an assembler. In this latter case, it lacks code. This code corresponds to missing object modules. Once this code is available, the last step is called static linking. It is carried out by a linker or a link editor. This tool is generally automatically called by the compiler. It makes it possible to bring together all of the code forming the application based on a format that depends on the operating system. This missing code is presented in the form of independent object modules or compiled functions belonging to static libraries (file extension .a and .lib depending on the OS). This linker resolves address correspondence problems. The executable file is called a.out by default in UNIX; under Windows, it has a .exe extension or, formerly, a .com extension. This file can then be executed by the microprocessor (MPU for MicroProcessor Unit). To do so, it is loaded by the OS and then given execution control. The loader allocates memory, initializes the environment and can resolve address correspondence problems if necessary. It is also responsible for launching the program. To conclude, note that there are link editors/loaders.

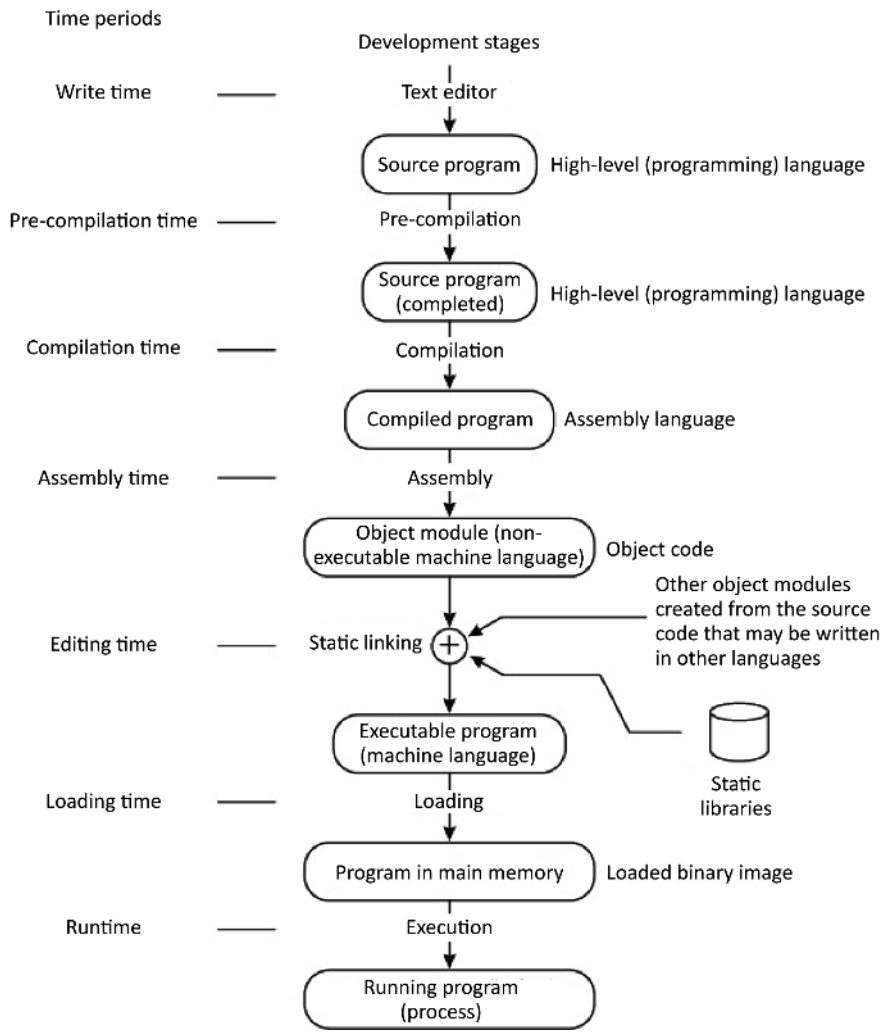

Figure 1.3. *Development chain for a program written in a compilable high-level language (here, C)*

There are two main language families that determine how a program is executed: interpreted and compiled languages. In the first case, the interpreter analyzes an instruction from the source program each time it is to be executed to determine how to do so, and this is done at the time of execution (Figure 1.4). In the second case, compilation and assembly translate a source program (written in a high-level programming language) into an object module. This takes place during compilation and assembly. Following link editing, the execution of an object program takes place during execution. Between the two categories, there are hybrid languages that are compiled and then interpreted (semi-compiled language) like Java. For the latter, the source is compiled to obtain instruction byte code in an intermediate language. These instructions are then interpreted by the virtual machine or, for faster execution, compiled on the fly. Another approach is the Forth language, which is both interpreted and compiled.

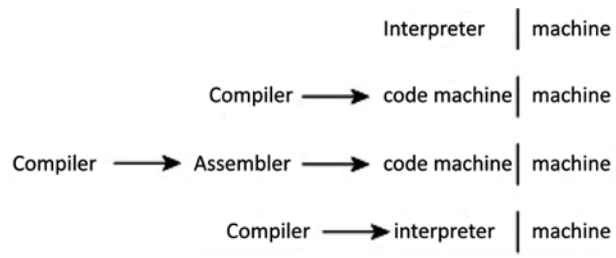

Figure 1.4. *Relationship between the type and levels of languages*

1.1.3. *Mixed-language programming*

Mixed-language programming consists of developing an application by using multiple languages. Programming is thus modular. A classic case is the combination of C, C++ and AL. The linker is responsible for creating the executable. It resolves reference problems (i.e. addresses) between different modules (Figure 1.5) by linking the symbolic words to the implementation addresses. This type of programming is complex and tends to generate errors. Each source is compiled with the specific language's compiler. At this stage, the tools become shared. Link editing brings together the various object modules to generate the application.

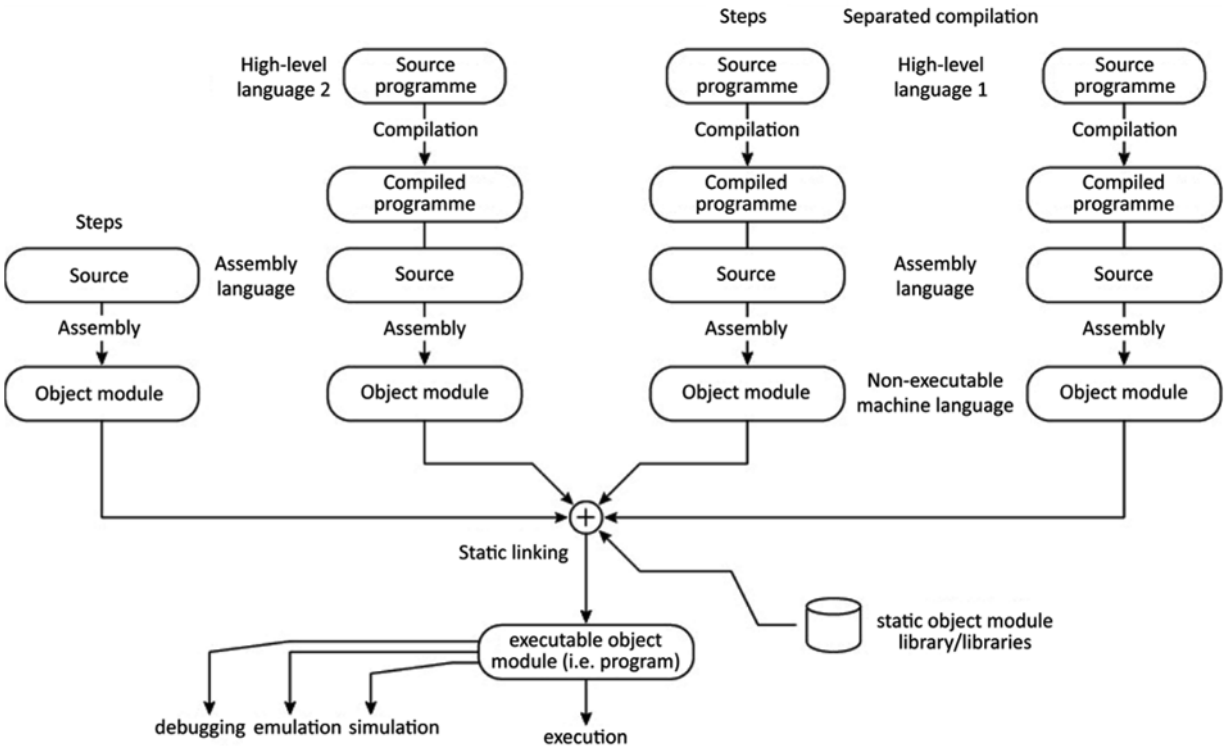

Figure 1.5. *Software development chain in a mixed-language environment and with separated compilation*

1.1.4. *Compatibility and software interfaces*

The concept of software compatibility is presented in § V4-3.3.2. It is relevant at three levels of the software development chain: source,[2] object and machine code (Figure 1.3).

Compatibility at the source code level today refers to high-level languages because it is better for obvious reasons than assembly language. These languages offer high-level software functionality (data type structures, control structures, paradigms, etc.).

Compatibility at the object code level makes it possible to distribute the program without supplying the source code. It is then necessary to carry out link editing (*cf.* § 1.2.2) with system libraries on the host computer, an example being libc in the C language. The specification takes over from the source code with, in addition, the definition of an object file format such as COFF or ELF (respectively Common Object File Format and Executable and Linkable Format, *cf.* § 1.2.2 for more details).

Compatibility at the machine code level or binary compatibility allows an application to be directly executed (ready-to-run). It requires the definition of symbolic data types (compatible with the high-level language declarations such as the long and int types in C, for example) and the specification of the alignment of structures and data. The other definitions are sections (code, data, stack and heap), their maximum sizes and the memory map (location), and the linker, which generates absolute addresses before knowing them. An example specification is BCS (Binary Compatibility Standard, Anderson *et al*. 1989) for Motorola's M88000 RISC (Reduced Instruction Set Computer) processor (this will be covered in a future book by the author on microprocessors). It is not necessary to compile or edit links for the application before execution (ready-to-run application). It thus provides independence from the programming language used.

These types of compatibility require standard interfaces. An interface is a bridge between two software layers or between a software and a hardware[3] layer. It provides an abstraction of the lower layer. It was necessary to develop a standard binary interface at the OS level (called the system interface), which is done symbolically, either at the programming language level with an API (Application

2 Or programs.

3 The hardware layers and their interfaces will be covered in a future book by the author.

Programming Interface) or at the binary level with an ABI (Application Binary Interface), as illustrated in Figure 1.6. A Hardware Abstraction Layer (HAL) makes it possible to detach the OS from the hardware implementation. ISA was described in § V1-3.5.

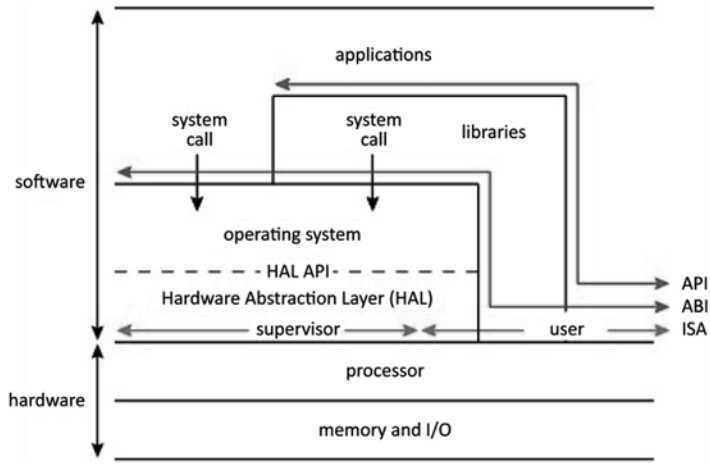

Figure 1.6. *API and ABI interfaces and Hardware Abstraction Layer. For a color version of this figure, see www.iste.co.uk/darche/microprocessor5.zip*

An API or application programming interface enables compatibility at the source level language in almost all cases for a high-level language (the C language, for example). It enables portability of applications at the source level (*cf.* § V4-3.2.3). It offers a set of services while masking the implementation details. The API is an abstraction for service calls. The invocation of these services is done in the form of standard or non-standard library functions. In the simplest case, the function is a simple wrapper that adapts/translates the high-level conventions into those of the lower-level. In the most complex cases, there are several system calls and the called function is processed. An example is SVID (System V Interface Definition) which defines the C language programming interface with UNIX System V (Novell 1995a, 1995b, 1995c). Another example is the IEEE 1003.1™-2013 – POSIX standard (IEEE 2013). This specification defines a set of executable functions, a standard library (library API) and a system API. A change in the API requires the application to be recompiled. The API belongs to the Abstract Machine Interface (AMI) located between applications and the OS. The latter also specifies the allowed instructions and the memory access model.

The ABI is a set of specifications to which the executable must conform so that it can be executed in a given execution environment. It defines a standard binary interface to be able to execute the compiled application. It is located between the applicable program and the OS, a library, or a set of I/O routines such as the BIOS (Basic Input/Output System, *cf.* § 4.2.2 in Darche (2003) and § 3.5.3). It enables portability of applications at the binary level. We can consider the ABI as equivalent to the API at the object code level. An example is System V (SCO 1996a, 1996b, 1997). We should also mention iBCS (Intel Binary Compatibility Standard), which allows the same binary code to function on x86 platforms under different Unix systems. It is generally made up of two parts, one that is common for all architectures and one that is specific to a particular architecture. It specifies a set of functions that the OS provides to the program user as well as how they should be called. The ABI instructions belong to the user set, not the system. The transfer of control to the OS is done through the intermediary of software interruption, which can be compared to a function call with (potential) passing of parameters under constraint. The potential parameters are generally passed by registers, or less commonly on the stack. The ABI therefore includes low-level information specific to the target architecture. This is the machine interface, the function call sequence and the interface with the OS. The machine interface describes the underlying architecture (i.e. ISA) and the data representation. The data representation specification provides its types with the format, order of storage (endianness; Cohen 1981, *cf.* § 2.6.2 in Darche (2012)) and associated alignments. The detail of the call sequence includes the register usage conventions, the stack frame layout convention and the passing of parameters (number, passing mode and type). The interface with the OS describes the exception interface, signal management, virtual addressing, process initialization (stack, registers, etc.) and debugging support. The executable and object file format are also specified (header, sections, etc.) as well as a library format. The ABI secondarily specifies the alphanumeric code used, for example, for character-based data control (\n, for example), or for the package data file. It provides information about loading the program and about potential dynamic linking. The ABI must support the API's libraries. Tools such as the compiler, the assembler and the linker, that is, the development chain, make reference to the ABI. An EABI (Embedded ABI) refers to the ABI version used in embedded systems. Its specific characteristics are the absence of a linker and modified memory management.

The interface between the OS and the hardware, the latter of which is optional, is called the Hardware Abstraction Layer. It is a software layer in the OS that abstracts the underlying hardware and is accessible via an API. An example is Windows. Drivers and hardware-specific software such as hot-swap management belong to this layer. It enables the addition of new hardware without having to change the OS's programs. Real-Time Operating Systems (RTOS) may not use a kernel (kernel-less approach), but instead a library linked to the application (library-based RTOS).

Using a standard API or ABI enables software compatibility (*cf.* § V4-3.3.2). The former enables application portability at the source-language level, and the latter does so at the binary level.

It should be noted that the idea of the machine is relative. As specified by Smith and Nair (2000), it depends on the point of view under consideration. As part of an operating system, the machine is the hardware support enabling execution, and the software interface is the ISA. As part of a process executing a program for a user, the machine is the OS, which supplies storage and memory as well as services such as I/O, and hardware support for execution is the ABI.

1.2. Fundamental software tools for development

The three primary tools required to develop an application are the assembler, the linker and the loader/launcher; there is also a tool called a disassembler. "Tuning and testing" will be addressed in the following chapter.

1.2.1. *Assembler*

The first assemblers were assembler-launchers (assemble-go system). They were responsible for loading subprograms written in assembly language into memory by translating binary instructions on the fly (i.e. a single program read), to call addresses[4] from the main program and to launch execution. In its modern form, it is a language translator. The first assemblers in modern form were SOAP (Symbolic Optimizer and Assembly Program) on the IBM 650 (Poley and Mitchell 1956) and then SAP[5] (SHARE Assembly Program) on the IBM 704 (Wegner 1976). A description of the latter can be found in Helwig (1975). Two representatives from the PC-Wintel[6] world are MASM (Microsoft Macro Assembler) and Turbo-Assembler® (tasm) from Borland.

The assembly of a source file will give rise to the creation of an output file named "listing" (the extension for the generated file is .lst), an object module, either absolute or relocatable, and a list of cross references (Figure 1.7). It should be noted that comments are deleted, which is handled by the precompiler (preprocessor) for a high-level language.

---

4 The term "address" refers to the logical address, a term belonging to the concept of Virtual Memory (VM, this will be covered in a future book by the author on storage). If there is no logical translation, it is a Physical Address (PA).

5 The original definition from IBM was *Symbolic Assembly Program*.

6 A contraction of the two names Windows and Intel.

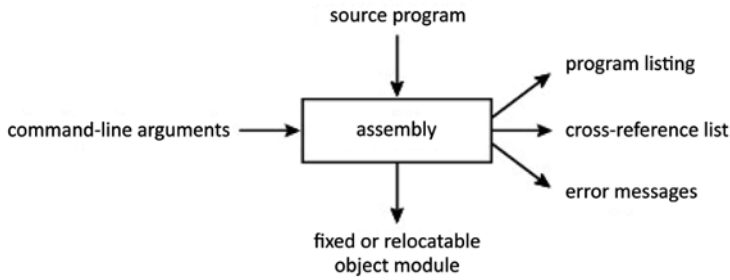

Figure 1.7. *Information flow for assembly*

Given that this is a symbolic language, translation is more than transliteration, hence the term assembly (Wegner 1968). Assembly can be broken down into two parts, analysis and synthesis. The latter can be divided into phases or steps (Figure 1.8). For a given language, syntax and semantics must be defined. Syntax concerns the formal aspects of a language. It manages the organization of instructions and data. Semantics concerns the meaning of instructions. Assembly will therefore begin with a lexical or lexicographic analysis. Its function is to pick out, from the data flow, a series of symbols called lexical or syntactical units, lexemes or tokens, from whence its other name, the scanner or tokenizer, or linear analysis, because the source is read from left to right, line by line. The typographic space is the base separator. There exist predefined lexical units in the language such as reserved words, operators, register names and separators (e.g. :, {, }, ;, etc.), as well as others defined by the user, identifiers[7] and numerical and alphanumeric literals (i.e. character strings). A second stage consists of determining to which family of units these lexemes belong. This is the sort phase carried out by the analysis. A data structure called a "symbol table" was initialized during these first two phases. This table will contain all of the elements to which the analyzer will add information such as their name, username, unit type such as identifier, constant (also called literal), language keyword or comment. These will also be initialized if applicable. An identifier is the name of a variable, constant, subprogram or label. It will then be elaborated during the following steps. From the point of view of implementation, these two functionally distinct steps are united (Wilhelm and Maurer 1994). This is followed by syntactic analysis, also called hierarchical analysis. It is carried out by the analyzer with the same descriptor (*parser*). It consists of verifying whether the source is respecting the grammar of the language. A grammar defines the rules for constructing sentences in the language, that is, lines of code containing declarations and instructions. One of the formal systems for

7 An identifier or username is a noun (*cf*. a character string) designating, among other things, a variable, a constant, a subprogram or a label.

describing a grammar is the Backus–Naur Form (BNF, Backus *et al*. 1960, 1963; Knuth 1964; ISO 1996). During this step, syntax errors are addressed. Semantic analysis is the penultimate step. It verifies, among other things, whether the variable format is coherent between registers and/or variables. Code generation is the final step. Code generation will involve symbolic evaluation. It associates an operation code or opcode with each instruction and, to each identifier, an address, when this correspondence is possible. If not, the linker stops associating to generate the final executable. The mode of assembly can be absolute or relative. This refers to the addressing mode (*cf*. § V4-1.2) used to reference the identifiers. Relative addressing enables address translation (relocation, *cf*. § V4-3.1.4), which makes it possible to install the code and the variables anywhere in memory without having to recalculate the addresses.

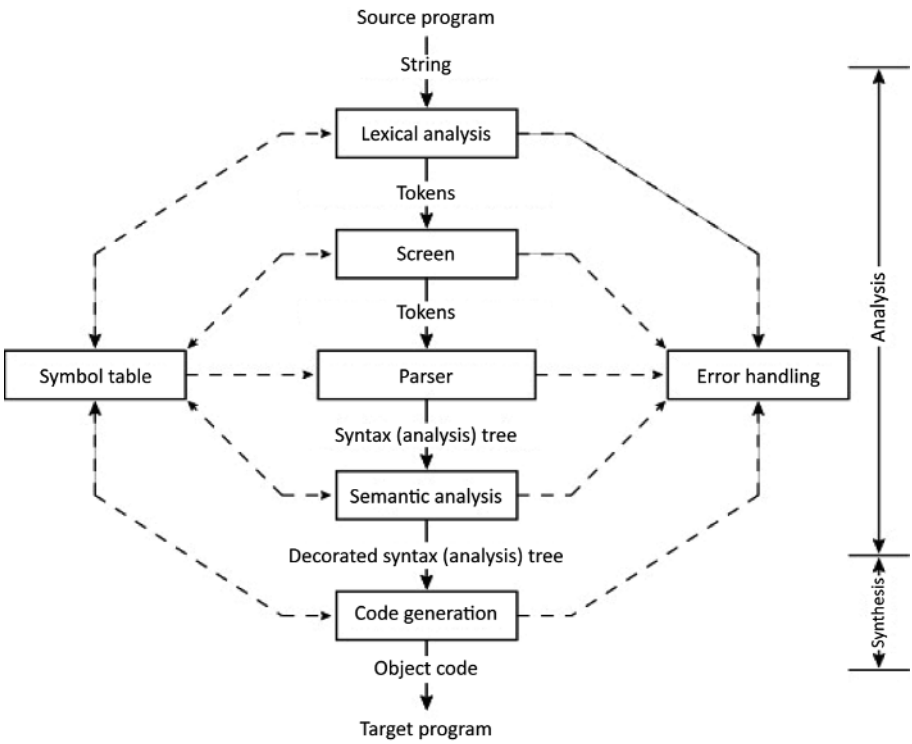

Figure 1.8. *Functional phases of assembly*

The structure of the symbol table will change depending on the step in the development chain. During assembly (assembly time), this data structure will contain, in its most complete form, the lexical units and, in less complete forms,

only identifiers with their attributes. When included in an object module, it makes it possible to resolve references during linking time by connecting the identifier and the reference. During execution, this table, which can be optionally included, enables the debugger to know the names of the identifiers. Figure 1.9 shows an example of this kind of table, extracted from the listing file in Figure 1.10.

```
Turbo Assembler  Version 3.1        06/10/11 17:32:02       Page 2
Symbol Table


Symbol Name                          Type   Value

??DATE                               Text   "06/10/11"
??FILENAME                           Text   "div     "
??TIME                               Text   "17:32:02"
??VERSION                            Number 030A
@32BIT                               Text   0
@CODE                                Text   DGROUP
@CODESIZE                            Text   0
@CPU                                 Text   0101H
@CURSEG                              Text   _TEXT
@DATA                                Text   DGROUP
@DATASIZE                            Text   0
@FILENAME                            Text   DIV
@INTERFACE                           Text   00H
@MODEL                               Text   1
@STACK                               Text   DGROUP
@WORDSIZE                            Text   2
BOUCLE                               Near   DGROUP:0012
D                                    Word   DGROUP:0002
DEBUT                                Near   DGROUP:0000
DIV0                                 Near   DGROUP:0025
FIN                                  Near   DGROUP:002C
FIN_MES                              Byte   DGROUP:0024
MESSAGE                              Byte   DGROUP:0008
N                                    Word   DGROUP:0000
OP                                   Near   DGROUP:000D
Q                                    Word   DGROUP:0004
R                                    Word   DGROUP:0006

Groups & Segments                    Bit Size Align  Combine Class

DGROUP                               Group
  STACK                              16  0100 Para   Stack   STACK
  _DATA                              16  0025 Word   Public  DATA
  _TEXT                              16  0030 Word   Public  CODE
```

Figure 1.9. *Example of a symbol table extracted from the listing file in Figure 1.10*

As with high-level languages, assembly can be conditional (CA for Conditional Assembly), which makes it possible to assemble only some parts of the program. An example use case is the potential to choose whether to include instructions from an application update. As with a compiler, the assembler, to carry out the translation, generally performs two passes through the source code. A mono-pass assembler directly translates symbolic references. But there are cases when this translation cannot be done, particularly when there is a forward reference for a variable or a label. There are therefore "two-pass," or even multi-pass assemblers. Assembly time is a function of the number of passes, of course.

The structure of the generated file, that is, the object module, can be broken down into a general information header, code and data, the area for useful address translation during a change of address, the global symbol table and the optional debugging information. The object module can be called absolute if the address of the program start was fixed and all of the identifiers' addresses are generated from it (this is the case in a mono-pass assembler). The memory dump therefore cannot be moved to main memory, contrary to a relocatable module. Modern assemblers generate this second kind of object module.

At the programmer's request, a listing file can be generated. The data line for this type of file (Figure 1.10) is typically divided into three areas called fields that are, from left to right, the line number (expressed in base-10 most of the time), the memory address (expressed in hexadecimal), the size, the variable name (optional) and the initialization value (optional), which corresponds to the assembly of the fourth zone, the source, as described in § 1.3.

The instruction line is typically divided into three areas called fields that are, from left to right, the line number (expressed in base-10 most of the time), the memory address (expressed in hexadecimal[8]), the machine code, then the source. A cross reference table can be constructed from the symbol table during assembly. For each symbol, whether internal to the program or external (i.e. public), there will be an entry in this table with its name, its value and the list of instructions it references, or even the number of the corresponding line. This can be seen in this file. This list facilitates debugging, among other things, and it can be used by a compiler.

8 This foundation enables a more compact representation of values compared to NBC (Natural Binary Code).

```
Turbo Assembler  Version 3.1        06/10/11 17:32:02       Page 1
div.asm


                                ; nom du programme : div.asm
                                ; division par additions successives
                                ; une approche simple

                                        IDEAL
                                        DOSSEG
 0000                                   MODEL TINY
                                        P8086

 0000                                   STACK 100h

 0100                                   DATASEG
 0000  000C                     N       DW 12
 0002  000C                     D       DW 12
 0004  0000                     Q       DW 0
 0006  0000                     R       DW 0

 0008  44 69 76 69 73 69 6F+    message DB "Division par zéro impossible."
       6E 20 70 61 72 20 7A+
       82 72 6F 20 69 6D 70+
       6F 73 73 69 62 6C 65
 0024  24                       fin_mes DB "$"

 0025                                   CODESEG

 0000  B8 0000s                 debut:  mov ax,@data
 0003  8E D8                            mov ds,ax
 0005  8E C0                            mov es,ax

 0007  8B 0E 0002r                      mov cx,[D]
 000B  E3 18                            jcxz div0

 000D                           op:
 000D  A1 0000r                         mov ax,[N]
 0010  33 DB                            xor bx,bx

 0012                           boucle:
 0012  43                               inc bx
 0013  8B D0                            mov dx,ax ; sauvegarde temporaire du reste
 0015  2B C1                            sub ax,cx
 0017  73 F9                            jnc boucle

 0019  4B                               dec bx
 001A  89 1E 0004r                      mov [Q],bx
 001E  89 16 0006r                      mov [R],dx
 0022  EB 08 90                         jmp fin

 0025                           div0:
 0025  BA 0008r                         mov dx,offset message
 0028  B4 09                            mov ah,9
 002A  CD 21                            int 21h

 002C                           fin:
 002C  B4 4C                            mov ah,4ch
 002E  CD 21                            int 21h
                                        END debut
```

Figure 1.10. *Example of a listing file generated by tasm (symbol table in Figure 1.9)*

There are different types of assemblers. A macro-assembler is an evolution from traditional assemblers that enables the creation of macro-instructions (*cf.* § 1.3.4), which, once substituted, add to the code. This makes it possible to simplifying how a program is written. A cross-assembler generates code for a different processor than the one executing the assembler, as opposed to a naive or resident assembler, also called a self-assembler. This is required when the target on which the application

will be executed is not sufficiently powerful (i.e. MPU power and memory and storage capacities) to run the development tools. It is often used to develop embedded applications. A multiprocessor assembler is capable of generating code for microprocessors in a different family. There is also something called a meta-assembler, a scientific curiosity that makes it possible to write an assembler as they were initially described (Ferguson 1966). A High-Level Assembler (HLA) approaches high-level languages by enabling variable types and using control structures and macro-instructions. One of the first was described by Wirth (1968) with his PL360 language for the IBM System/360. A modern version of this type of language is HLA, proposed by Hyde (2010). Some high-level language compilers, such as the one for C, make it possible to insert lines of assembly language. This is referred to as an inline assembler. This is useful for directly accessing the MPU's instruction set or for calls to the OS. The downside is the loss of portability since this language is specific to an architecture or a specific MPU. Finally, we can mention the patch assembler, which is used in debuggers to modify application code in real-time to correct an error and test it immediately.

1.2.2. *Linker*

During the last step before obtaining the program executable, the linker (linkage editor) or link editor provides address translation functions (relocation, *cf.* § V4-3.1.4) and symbolic resolution. It selects the implementation address for the code and the variables as a function of the memory model, for example. It defines a size as an assembler and a start address (TOS for Top Of Stack) for the stack. If all of the modules are presented, it resolves the address correspondence if necessary. Generally, from the object modules and static and dynamic libraries (or DLL, for Dynamic Link Library), that is, shared between several applications, and following the arguments entered on the command line, it generates a module, either an executable or a relocatable, as illustrated in Figure 1.11. The executable file has an extension of .com, .exe (Windows environment), or an execution permission[9] (x for UNIX), or is a file for programming an EPROM (file extension .hex, for example). The second case, in which a new relocatable object module is generated, is due to the fact that the linker does not continue to the end, that is, that there remain unresolved references because, for example, of a missing library. Symbolic information generated optionally during assembly can be supplied as well for debugging at the executable source level (traditional file extension .map). This information can be related to sections (number, type, start address and size) and

9 It is no longer a question of having run permissions because a program can be executed or interpreted.

identifiers (name, type, address, allocation class, etc.). A control file (LCF for Linker Control File), also called a linker script file, contains the instruction to organize these sections and to define memory locations. As during compilation and assembly, there exists the notion of a pass or passes.

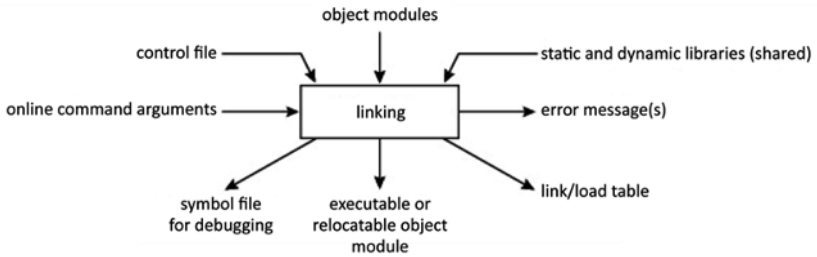

Figure 1.11. *I/O for link editing*

A library is a collection of general-use object modules (i.e. compiled functions) collected by domain or subject (e.g. the standard language library, a graphics library, etc.) that is intended for reuse. This idea of a library is more present in high-level languages like C with, for example, its standard library, than in AL. That said, there was an attempt to offer a standard library by the University of California Riverside named the UCR Standard Assembly Language Library, for use with the 80x86 family. There are three types: static, dynamic and import.

A static library or archive contains relocatable object modules. At link time, the object module's code is included in the output file, which is its major disadvantage for large applications.

A dynamic link library (file extension .dll on Windows), also referred to as a shared library (hence its file extension .so for shared object under UNIX) is a library of compile functions shared between applications that, instead of being linked definitively to the application code during static linking, are stored in memory upon execution. A distinction can therefore be made between dynamic linking and dynamic loading, as illustrated in Figure 1.12. Dynamic linking takes place when the application is launched. Dynamic loading takes place on demand when called, therefore under the program's control. Linking is delayed at execution (runtime) in both cases and is therefore done dynamically. Postponing the loading of a shared library until it is called allows for an increase in available memory space and time saving for the initial launching of the application. This latter case is called lazy linking. This approach favors modularity and code re-use. Under Windows OS, we find libraries in the form of files with extensions such as .dll, .ocx, .cpl or .drv. Under Unix, libraries carry the extension .so (shared object).

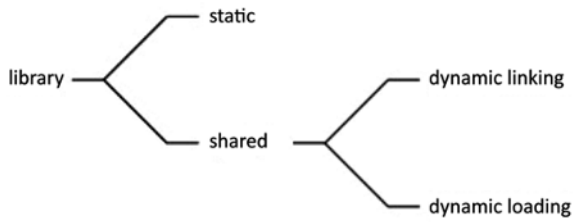

Figure 1.12. *Types of libraries*

An import library is a hybrid that enables management of loading and use of the shared library. Under Windows, it occurs in the form of a static library with the same name as the dynamic library and is statically linked to the application.

The generated file format is generally dependent on the OS. The format a.out, originally a UNIX format, is made up of a header, the code or text section, the data section and other sections. The COFF format, which originated on UNIX System V, is a format for executable files, objects and libraries introduced in UNIX System V to replace the a.out format. It was adopted by Microsoft. This format has itself been replaced on most systems by ELF, with the notable exceptions of the DJGPP compiler for object files and of Microsoft Windows, whose Portable Executable (PE) format is a modified version. This is a much more complex format because it can be composed of an arbitrary number of sections, such as, for example, under Linux, with .text for instructions, .data for initialized variables, and .bss for uninitialized variables, .rodata for constants, .symtab for the symbol table, etc. There are other formats like those in the PC (Personal Computer) world. We can cite Intel's OMF (Object Module Format) and Microsoft's PE format, which is derived from COFF. The latter is the executable file format (file extensions: .exe and .ocx – ActiveX and OLE) and dynamic files for 32-bit Windows operating systems.

To manage a library, there is an archiver/librarian. The main functions of this archiver are insertion at any position, extraction, displacement, replacement and deletion of an object module in a file named library in the standard format (COFF, for example). It also lists its contents. Under Unix, we can mention examples of archivers such as `ar` and `ranlib`.

For the sake of completeness, the linker uses a binding form of linking.

1.2.3. *Loader/launcher*

The loader transfers a program that is to be associated with a process from mass storage to main memory. In the early days of programming, it was associated with

the assembler (assemble-go-loader) and the linker. In the simplest case, it consists of a simple copy operation. Historically, loading was done at a fixed address. Today, for the sake of memory management, the load address is variable. Other functions can be added such as memory space allocation.

Loading the OS is a special case. It is carried out by the startup program called BIOS (*cf.* § 4.2.2 in Darche (2003) and § 3.5.3). This firmware contains a program called a primary boot loader that is responsible for loading and launching the secondary boot loader at sector 1 of track 0 (cylinder 0 head 0, more precisely) (MBR or Master Boot Record) or a launcher such as grub for Linux. It should be noted that there also exists network launching, or boot by LAN (Local Area Network).

Putting this software into ROM makes it impossible to add new support for peripherals. This is why BIOS has evolved towards UEFI (Unified Extensible Firmware Interface, *cf.* § 3.5.3), an initiative begun by Intel.

1.2.4. *Disassembler*

This section cannot conclude without addressing the disassembler, which is the flip-side of the assembler. The idea is to be able to reconstruct the source from the object module. This can be necessary, for example, if the source was lost inadvertently or for the purpose of reverse engineering. It analyzes the machine code to retrieve the mnemonics. But recovering data structures and comments is problematic. Effectively, the tool has no information to recover symbolic names. This can be done by integrating information during assembly that is useful for disassembly and debugging, such as a symbol table or the compiler name. To do so, compilation and assembly must be done with the necessary options to include this information. We can distinguish between traditional disassembly of a program's machine code and the kind that comes from traces (*cf.* § 2.2).

1.3. Assembly language

Assembly language is a symbolic representation of machine-executable binary instructions. Table 1.1 shows a comparison between high-level (HLL) and low-level (programming) languages with regard to size and development time. Jones (1998) also compares this language to COBOL (COmmon Business Oriented Language) by adding the criteria of function points (Albrecht 1979). We see that assembly language provides more code and that the development time is greater, at least double in relation to C. On the other hand, AL provides better productivity because the generated code is closer to the machine. But application maintenance is more difficult than with HLL.

Languages	Size (in lines of code)	Development effort (in man/months)
Assembly	10,000	40
Macro-Assembly	6,666	28
C	5,000	22
COBOL	3,333	16
Pascal	2,500	13
Ada	2,222	12
Basic	2,000	10

Table 1.1. *Language comparison (based on Jones (1986))*

Why use assembly language? Originally, it was used to avoid the need to directly work with binary code. By providing, among other things, symbolic notation for identifiers and instructions, assembly language offers an ease of use that machine language, which works with 1s and 0s, cannot. AL is the generable and readable version of machine language. Symbolic names enable accelerated development and decrease the number of programming errors. It makes it possible to eliminate syntactic errors (misuse, missing instruction, bad machine code, etc.), which will be detected during the assembly phase (*cf.* § 1.2.1). Commands facilitate programming. With the invention of high-level languages, assembly language has been progressively restricted to niches where the increase in execution speed and memory space is a priority. Its instructions are those from the MPU's instruction set. A program written in assembly language is therefore more efficient in terms of speed and memory usage than a program written in a high-level language. Examples include time-critical software functions such as schedulers and low-level I/O drivers for a real-time operating system or a loader. Originally, MPU evaluation kits (*cf.* § 2.1.1) only contained a few kibibytes (KiB) of memory! Embedded and on-board systems require even more economic use of memory space. Working near the processor makes it possible to optimize memory space and achieve compact code. Microcontrollers, components that even now contain limited memory, are still programmed using AL. Furthermore, a microprocessor that is not in widespread use may not have a compiler for a given language. AL is the only recourse in this case.

Assembly language is of pedagogical interest to help students understand in detail the operation of the MPU, as well as some of the mechanisms of high-level languages. Without detailed knowledge of MPU programming, it is not possible to understand the internal operation of this component. Superscalar architecture (this

will be covered in a future book by the author on microprocessors), for example, cannot be understood without knowledge of the idea of branching or register addressing. It can also provide an introduction to learning about architecture. From the software perspective, it is also possible to study concepts such as function calls and passing parameters, control structures in high-level languages, and, thus, possibly, to learn how to program more efficiently through knowledge of how the compiler will translate the program's source code. Operating system mechanisms such as system calls are easily explained with AL. It is also possible, in order to not derail the student, to program in the style of a high-level language (Crookes 1983).

But today, operating systems include several tens of millions of lines of code. Programming in assembly language is no longer humanly possible. On the other hand, it can also be used to create specific interfaces not provided by the high-level language. For example, it is possible to use functions written in another language using an interface written in AL. High-level languages or an operating system may not provide the primitives necessary to precisely manage an "exotic" and therefore non-standard programmable component. Calls can be made specifically, for example, with the help of a specialized instruction or by using a register. This is particularly true for I/O. It is also possible to add primitives from an operating system in a high-level language. It also enables the production of specific interfaces not provided by the language, such as a peripheral driver. AL provides complete access to the MPU's instruction set, for example those that process multimedia (*cf.* § V4-2.7.1). This allows a programmer using a high-level language to access hardware using the inline assembler more efficiently. Embedded systems are another area. These are generally programmed using several types of language, one or more high-level languages for high-level applications, and AL for interactions between the operating system and the hardware (storage and I/O). Finally, the link editor, such as the one in Linux, can call a module written in AL to generate an executable.

It is important to note that memory is less and less expensive. Compilers for high-level languages (C, C++, Pascal, Ada, etc.) generate increasingly optimized code when evaluated based on criteria including runtime and storage/memory use. Furthermore, increasingly powerful microprocessors can undergo a loss of power because of non-optimized code generated by a compiler. Generation of code in assembly language can only be automated. Another major inconvenience is that this language is not portable or provides limited portability, limited to a family of MPUs. Inserting code inline in AL therefore eliminates the high-level language's portability (*cf.* § V4-3.2.3). A program therefore has to be rewritten for each microprocessor (instruction or equivalent block of instructions). Memory alignment (*cf.* § 2.6.1 in Darche (2012)), if the insertion is done directly to the machine code, can lead to a decrease in performance (*cf.* § V4-3.4). Upward compatibility (*cf.* § V4-3.3.3) of code, as is the case with the x86 family, is useful, but requires the designer to make some of the architecture inalterable or to add complexity. Furthermore, this language

is more difficult to learn compared to a high-level language. Programs are more difficult and take longer to write, read, understand, debug and maintain.

From the architecture perspective, modern approaches such as VLIW (Very Long Instruction Word, this will be covered in a future book by the author on microprocessors) or speculative execution make it difficult to use AL because their complexity is offloaded to the compiler. Even if AL programming is still possible, it does not enable the optimal use of the underlying architecture.

1.3.1. *Software development methodology*

As for any language, an analysis phase for the problem to be addressed is necessary before any program is written. A processing algorithm is described textually with the help of a pseudo-language or designed using a flow chart (the use of which is in the process of dying out), which is a logical representation of the algorithm followed by a flow chart specific to the machine. Then, the program can be written using a text editor, at first in text mode (originally, now abandoned), initially a line editor and now a full-page editor (modern form).

1.3.2. *Standardization of assembly language*

Every designer of a processing unit or microprocessor proposes their own names for instructions and modes for addressing, which introduced confusion or even errors during design and programming. Furthermore, assembly language, because it is specialized for a processor or family of processors, is not portable. Attempts to propose a generic assembly language, for example by Nicoud (1976), proved vain. A standard was issued to attempt to resolve the problem. This was the IEEE 694-1985 standard (IEEE 1985).

1.3.3. *Structure of a program*

Regardless of the microprocessor and the assembler, programs in assembly language are similar from the structural and syntactic perspective. This section also presents a generic model for a program. A program is traditionally broken down into three sections: assembly commands, data and instructions, as shown in Figures 1.9/1.10 and 1.13. The separator is the space or the tab symbol, which added a predefined number of spaces.

```
/*
 * test_led_switch.asm
 *
 *  Created: 07/05/2014 16:48:49
 *  Author: Atmel
 */

 ;***** STK500 LEDS and SWITCH demonstration
.include <8515def.inc>
.def Temp =r16 ; Temporary register
.def Delay =r17 ; Delay variable 1
.def Delay2 =r18 ; Delay variable 2

;***** Initialization
RESET:
      ser Temp
      out DDRB,Temp ; Set PORTB to output

;**** Test input/output
LOOP:
      out PORTB,temp ; Update LEDS
      sbis PIND,0x00 ; If (Port D, pin0 == 0)
      inc Temp ; then count LEDS one down
      sbis PIND,0x01 ; If (Port D, pin1 == 0)

      dec Temp ; then count LEDS one up
      sbis PIND,0x02 ; If (Port D, pin2 == 0)
      ror Temp ; then rotate LEDS one right
      sbis PIND,0x03 ; If (Port D, pin3 == 0)
      rol Temp ; then rotate LEDS one left
      sbis PIND,0x04 ; If (Port D, pin4 == 0)
      com Temp ; then invert all LEDS
      sbis PIND,0x05 ; If (Port D, pin5 == 0)
      neg Temp ; then invert all LEDS and add 1
      sbis PIND,0x06 ; If (Port D, pin6 == 0)
      swap Temp ; then swap nibbles of LEDS

;**** Now wait a while to make LED changes visible.
DLY:
      dec Delay
      brne DLY
      dec Delay2
      brne DLY
      rjmp LOOP ; Repeat loop forever
```

Figure 1.13. *Example of a source file for a microcontroller from the Atmel AVR family*

An assembly directive or pseudo-operation (a word taken from the SAP assembler) can have two functions. The first is to direct the assembly or link editing by providing non-deductible data. This can include, for example, the target operating system definition or a specific syntax to be used. The second concerns data structure and initialization (data directives) and the structure of instructions. This includes the definition of variables and memory areas. A directive is therefore not translated into instructions for the machine. It is instead an instruction for directing assembly. It is

also called a pseudo-instruction. Here are some typical examples. The ORG directive makes it possible to specify the load address, also called the assembly address. The end of the program is indicated with the END directive. EQU makes it possible to define a symbolic constant. This directive assigns a numerical value to a symbolic name (similar to but not exactly the same as #define in C with a parameterless macro). It is also possible to define variables, with or without initialization. To do so, memory space must be reserved (using the RMB directive for Reserve Memory Byte) and initialized. A macro-instruction (*cf.* § 1.3.4) is predefined using the MACRO directive. The directive is sometimes preceded by a period. Some AL may provide file inclusion (.inc directive, for example), similar to what C permits with its #include directive.

The data section enables the definition of variables with or without names and the specification of their format, with initialization if necessary. Normally, it begins with DATA. The label field enables the definition of variable names. The symbolic name will be assigned the current address value during assembly. An alignment directive may also be present. The syntax of a line is therefore as follows:

```
[label]     format [value]   [; comment]
VAR1        DB 4             ; variable definition
                             ; initialized to 4
```

Unlike a high-level language, AL does not offer data typing as a standard feature, but it is possible to define a format (i.e. the number of bits or bytes). The numerical values should be expressed with a specified or implicit base, generally base-10. By using prefixes[10] or suffixes, whether normalized or not, B for Binary, D for Decimal, H for Hexadecimal and Q for Octal. It is therefore implied, since originally there were only natural numbers. The type, a whole number or integer, is only revealed by the indicators. With the evolution of hardware, this language today offers data formats including both natural numbers and integers, floating and fixed-point numbers, character (strings), and binary (bit-string format). For character strings, the value is enclosed by double quotes and may include a directive that indicates the encoding (.ascii or .asciiz, for example). An example is MASM from Microsoft.

Under UNIX, the instructions section is called the text section. A line of instruction in a source file (Figures 1.9/1.10 and 1.13) are typically divided into four fields, which are, from left to right, the label, the instruction mnemonic, operator field, or operation with, if necessary, an operand field and, to conclude, the

10 For reference, the C language uses the prefixes 0x for hexadecimal, 0 for bytes and sometimes 0b for a binary pattern.

comments. We can therefore define its syntax and an example for the x86 family (Intel):

```
[label] [instruction] [operand[,operand]]       [; comment]
        mov ax,0
        mov cx,10
loop_begin:
        add ax,1            ; beginning of the loop
        loop loop_begin     ; end of the loop
```

The location or label field is filled with a character string, which makes it possible to find an instruction line, that is, a determined point in the code, for example, the destination of a jump. The label makes it possible to automatically calculate (i.e. with the assembler) the address of a jump, for example. To do so, the assembler maintains an up-to-date location counter that is incremented each time it passes to the next instruction by a value that depends on the length of the current instruction. This counter is reset to zero when the section changes, or it initialized to a defined value by a directive (ORG, for example). The definition of a label should begin at the beginning of a line and, potentially, should be terminated with the character ":". There is generally a label indicating the beginning of the program to the link editor (e.g. the label _start), which can also be called the entry point. The code field is filled with a mnemonic. The operand field contains between 0 and n-1 values or references depending on the addressing mode and the architecture. It is important to distinguish between an operand for an operation (mathematical definition) and, in assembly language, operands for an instruction, which could either be a "traditional" operand or a result. Depending on the style, the destination operand can be to the left,[11] as is the case for high-level languages (Intel or MPU DLX (DeLuXe) style), or to the right, since allocation is done from left to right (AT&T style, used in UNIX BSD (Berkeley Software Distribution)). The comment field is very important in AL source code, because the language is complex, and the source code is hard to read compared to high-level languages. It begins with the comment delimiter, which is the ";" character.

The address mode for the operand is indicated by the syntax. Square brackets typically indicate whether the mode is indexed or indirect. Table 1.2 specifies the standard conventions. The current address (location counter reference) is indicated by an asterisk (typographical symbol *) in the standard, or otherwise by the $ symbol. This makes it possible to calculate a jump address, for example.

11 This is called reverse ordering.

Address space operators	Symbols
literal value	#
direct address	/
register	.
relative address	$
direct page, base page, page zero	!
indexed or direct	[]

Table 1.2. *Address space operators in the IEEE 694-1985 standard*

In the IEEE 694-1985 standard, the working format for the instruction is indicated by adding, after a period, the mnemonic or, less commonly, its operand, following the standard in Table 1.3. A character string is initialized by putting the text in quotation marks. Typing for floating-point numbers is done in the same way, with FS for Short Floating Point, FL for Long Floating Point, FX for Extended Floating Point and FP for Packed decimal Floating Point.

Format (bit)	Mnemonic suffix	Equivalent
1	I, I1	
x	Ix, x natural number	
8	B	I8
16	S	I16, B2
32	L	I32, B4, S2
64	Q	I64, B8, S4, L2

Table 1.3. *Standardized format suffixes*

A comment, which is optional but strongly advised, enables documentation of a program to facilitate reading and maintenance of the source code. It is not taken into account during assembly. It begins with an agreed-upon character, in this case the semicolon. This beginning marker varies between manufacturers. It can also be @, #, *, or as is the case in C and C++., /* */ and //.

Finally, with regard to case, Intel, Microsoft (ASM86), Motorola, Zilog, Arm®, SGS and TI use capital letters in code. Keil follows the same convention, except for registers! Microchip, MIPS, IBM (Power architecture), Intel (IA-64 architecture)

and HP use lower-case letters in code. Atmel uses capital letters for documentation and lower-case letters in programming. Inmos puts mnemonics and registers in lower case in both documentation and programming.

1.3.4. *Macro-instructions*

The concept of a macro-instruction emerged from the EDSAC project in the term open subroutine (Wilkes *et al*. 1951). It was extended to high-level languages several years later (McIlroy 1960). A macro-instruction (for a character string substitution) or, in abbreviated form, a macro, is a symbolic name that can be substituted by a character string. It is also called a pseudo-instruction or synthetic instruction. This text can include directives, mnemonics or data declarations. To define one, a macro-definition must be used. They help avoid repetition in code, but are not functions, because the code is not factored. A macro enables the definition of new instructions or the use of higher-level constructions, close to those of high-level languages. In this way, it simplifies syntax and programming. A macro is customizable, which increases its expressive power. During assembly, formal parameters are replaced by actual parameters. A library of macro-instructions facilitates reuse. Here, we are approaching the syntax of a High-Level Language (HLL). A macro-assembler is an assembler that uses macro-instructions. During an assembly pass, the macro-instruction is expanded. Figure 1.14 shows an example macro followed by its expansion. Each language provides predefined macros to facilitate programming.

```
;  macro example
;  name: PUSH_REGS
;  function: stacking two registers
;  two parametres; register names

.macro PUST_REGS
     push @0
     push @1
.endmacros

utilisation:
     ldi        R18,0x00
     ldi        R17,0x02
     PUSH_REGS  R18, R17

expansion:
     ldi        R18,0x00
     ldi        R17,0x02
     push       R18     ;macro code
     push       R17     ;macro code
```

Figure 1.14. *Example of a program with macro-instruction and then expansion*

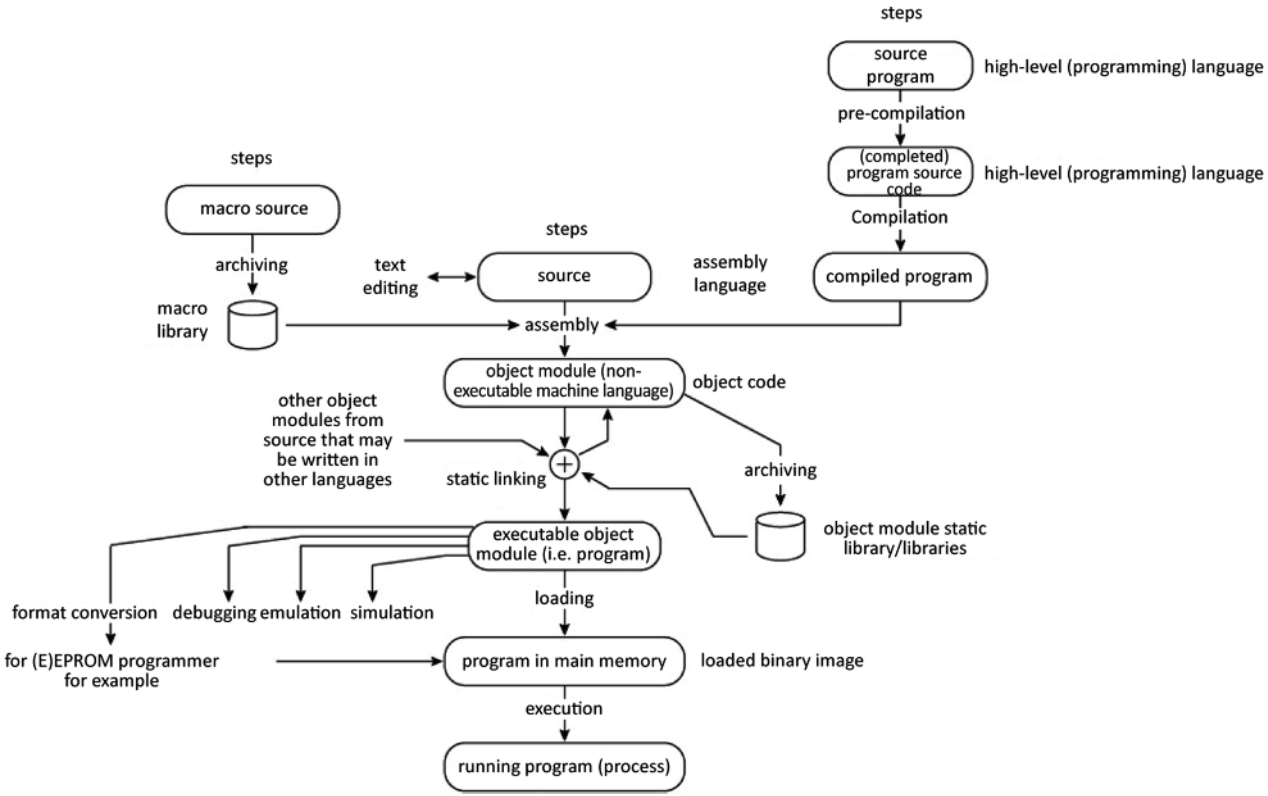

Figure 1.15. *Development chain with macro-management*

As for object module libraries, a macro-library manager (or archiver) exists with the same functions (*cf.* § 1.2). Figure 1.15 shows a summary of the development chain and tuning.

1.3.5. *Addressing*

Assembly language provides the programmer with six address spaces. These are the register space, stack space, heap space, text space, I/O space (*cf.* § V3-2.1.1.1 concerning the instructions `in` and `out`) and the control space.

Furthermore, there may be address modes specific to an assembly language, but which the microprocessor does not have. These are advanced modes that facilitate programming (*cf.* § V4-1.2.4.6).

1.4. Conclusion

The selection of a software development chain is important because it influences the application's development cost, performance and memory footprint.

The next chapter is focused on debugging and testing.