Programming embedded systems:

unit3

structure of embedded program

- There are many different approaches to development in embedded systems with the embed there is no software to install, and no extra development hardware needed for program download. All software tools are placed online so that you can compile and download wherever you have access to the Internet
- Notably, there is a C++ compiler and an extensive set of software libraries used to drive the peripherals

The embed Compiler and API

- One thing that makes the embed special is that it comes with an application programming interface (API). In brief, this is the set of programming building blocks, appearing as C++ utilities, which allow programs to be devised quickly and reliably. Therefore, we will be writing code in C or C++, but drawing on the features of the API.

- As just mentioned, the mbed development environment uses a C++ compiler.That means that all files will carry the .cpp (Cplusplus) extension. C, however, is a subset of C++, and is simpler to learn and apply. This is because it does not use the more advanced ‘object-oriented’ aspects of C++.

  In general, C code will C compile on a C++ compiler, but not the other way round. C is usually the language of choice for any embedded program of low or medium complexity, so will suit us well here. For simplicity, therefore, we aim to use only C in the programs we develop. It should be recognized, however, that the mbed API is written in C++ and uses the features of that language to the full. We will aim to outline any essential features when we come to the

- **Program Design and Structure**
  There are numerous challenges when tackling an embedded system design project. It is usually wise first to consider the software design structure, particularly with large and multi-functional projects. It is not possible to program all functionality into a single control loop, so the approach for breaking up code into understandable features should be well thought out. In particular, it helps to ensure that the following can be achieved:

- that code is readable, structured and documented
- that code can be tested for performance in a modular form
- that development reuses existing code utilities to keep development time short
- that code design supports multiple engineers working on a single project
- that future upgrades to code can be implemented efficiently.
-
  There are various C/C++ programming techniques that enable these design requirements to be considered, as discussed here, including: functions, flow charts, pseudocode and code reuse.

- **The role of Functions**
  A function is a portion of code within a larger program. The function performs a specific task and is relatively independent of the main code. Functions can be used to manipulate data; this is particularly useful if several similar data manipulations are required in the program. Data values can be input to the function and the function can return the result to the main program. Functions, therefore, are particularly useful for coding mathematical algorithms, look-up tables and data conversions, as well as control features that may operate on a number of different parallel data streams. It is also possible to use functions with no input or output data, simply to reduce code size and to improve readability of code. **Figure 6.1** illustrates a function call.

- There are several advantages when using functions. First, a function is written once and compiled into one area of memory, irrespective of the number of times that it is called from the main program, so program memory is reduced. Functions also allow clean and manageable code to be designed, allowing software to be well structured and readable at a number of levels of abstraction. The use of functions also enables the practice of modular coding, where teams of software engineers are often required to develop large and advanced applications. Writing code with functions therefore allows one engineer to develop a particular software feature, while another engineer may take responsibility for something else.

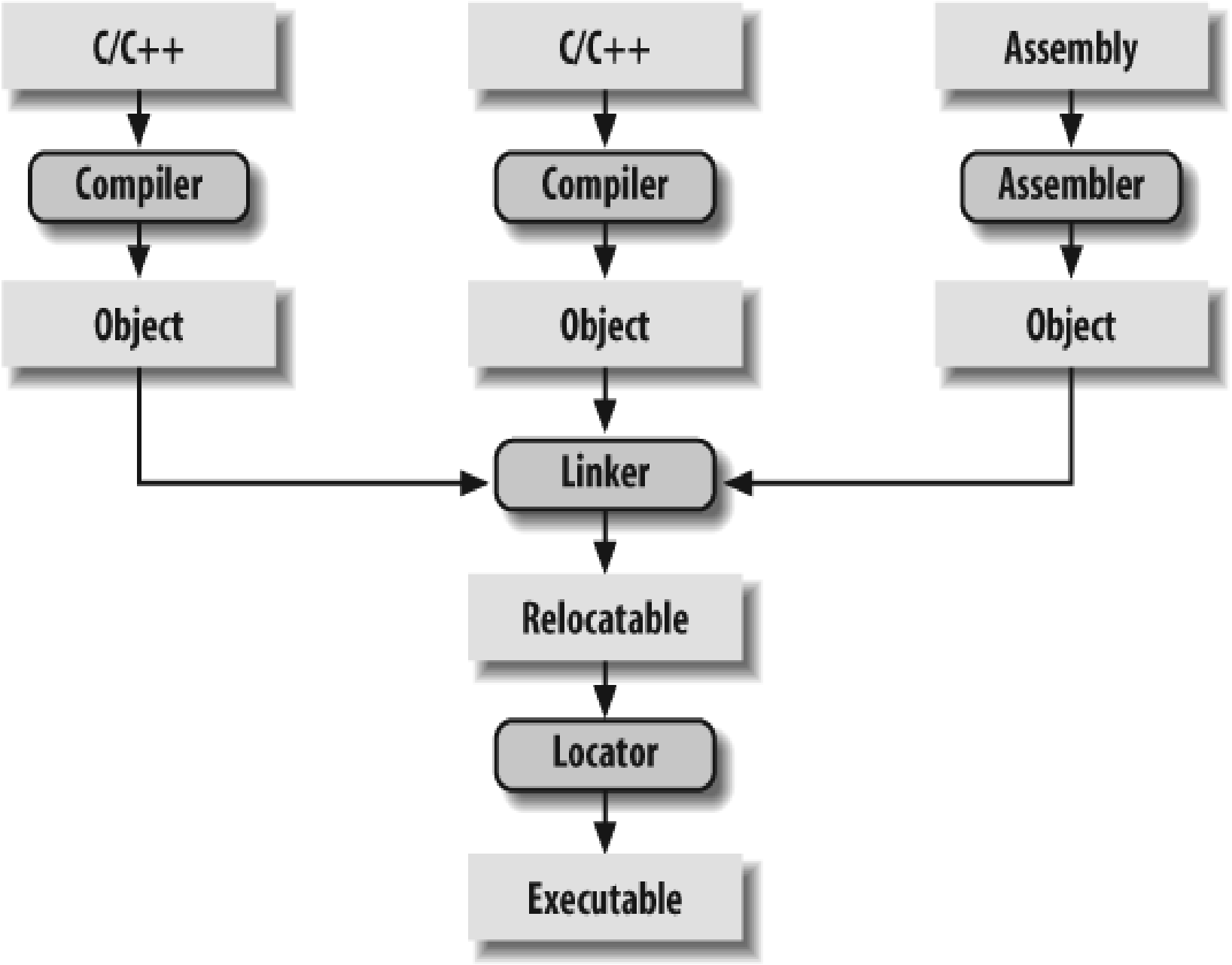
- Using functions is not always completely beneficial, however. There is a small execution time overhead in storing program position data and jumping and returning from the function, but this should only be an issue for consideration in the most time-critical systems. Furthermore, it is possible to ‘nest’ functions within functions, which can sometimes make software challenging to follow. A limitation of C functions is that only a single value can be returned from the function, and arrays of data cannot be passed to or from a function (only single-value variables can be used). Working with functions and modular techniques therefore requires a considered software structure to be designed and evaluated before programming is started.

Infinite loop

- A super loop is a program structure comprised of an infinite loop, with all the tasks of the system contained in that loop. Here is a general pseudocode for a superloop implementation:
- Function Main_Function() { Initialization(); Do_Forever { Check_Status(); Do_Calculations(); Output_Response(); } } We perform the initialization routines before we enter the super loop, because we only want to initialize the system once. Once the infinite loop begins, we don't want to reset the values, because we need to maintain persistent state in the embedded system.
- The loop is in fact a variant of the classic "batch processing" control flow: Read input, calculate some values, write out values. Do it until you run out of input data "cards". So, embedded systems software is not the only type of software which uses this kind of architecture.

The Build Process

- When build tools run on the same system as the program they produce, they can make a lot of assumptions about the system. This is typically not the case in embedded software development, where the build tools run on a *host* computer that differs from the *target* hardware platform. There are a lot of things that software development tools can do automatically when the target platform is well defined. [1] This automation is possible because the tools can exploit features of the hardware and operating system on which your program will execute. For example, if all of your programs will be executed on IBM-compatible PCs running Windows, your compiler can automate—and, therefore, hide from your view—certain aspects of the software build process. Embedded software development tools, on the other hand, can rarely make assumptions about the target platform. Instead, the user must provide some of her own knowledge of the system to the tools by giving them more explicit instructions.
- The process of converting the source code representation of your embedded software into an executable binary image involves three distinct ste


C/C++
C/C++
Assembly
Compiler
Compiler
Assembler
Object
Object
Object
Linker
Relocatable
Locator
Executable

Compiling

- The job of a *compiler* is mainly to translate programs written in some human-readable language into an equivalent set of opcodes for a particular processor. In that sense, an *assembler* is also a compiler (you might call it an “assembly language compiler”), but one that performs a much simpler one-to-one translation from one line of human-readable mnemonics to the equivalent opcode. Everything in this section applies equally to compilers and assemblers. Together these tools make up the first step of the embedded software build process.
- Of course, each processor has its own unique machine language, so you need to choose a compiler that produces programs for your specific target processor. In the embedded systems case, this compiler almost always runs on the host computer. It simply doesn’t make sense to execute the compiler on the embedded system itself. A compiler such as this—that runs on one computer platform and produces code for another—is called a *cross-compiler*. The use of a cross-compiler is one of the defining features of embedded software development.(in other wordA **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.)

Gcc compiler

- The job of a *compiler* is mainly to translate programs written in some human-readable language into an equivalent set of opcodes for a particular processor. In that sense, an *assembler* is also a compiler (you might call it an “assembly language compiler”), but one that performs a much simpler one-to-one translation from one line of human-readable mnemonics to the equivalent opcode. Everything in this section applies equally to compilers and assemblers. Together these tools make up the first step of the embedded software build process.
- Of course, each processor has its own unique machine language, so you need to choose a compiler that produces programs for your specific target processor. In the embedded systems case, this compiler almost always runs on the host computer. It simply doesn’t make sense to execute the compiler on the embedded system itself. A compiler such as this—that runs on one computer platform and produces code for another—is called a *cross-compiler*. The use of a cross-compiler is one of the defining features of embedded software development.