SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

19ECT312 – EMBEDDED SYSTEM DESIGN

III YEAR/ VI SEMESTER

UNIT 3 : PROGRAMMING CONCEPTS AND EMBEDDED PROGRAMMING IN C++

TOPIC 3.5 C++ PROGRAM COMPILERS

C++ PROGRAM COMPILERS

The Build Process

The process of converting the source code representation of your embedded software into an executable binary image involves three distinct steps:

1. Each of the source files must be compiled or assembled into an object file.

2. All of the object files that result from the first step must be linked together to produce a single object file, called the relocatable program.

3. Physical memory addresses must be assigned to the relative offsets within the relocatable program in a process called relocation.

C++ PROGRAM COMPILERS

The result of the final step is a file containing an executable binary image that is ready to run on the embedded system.

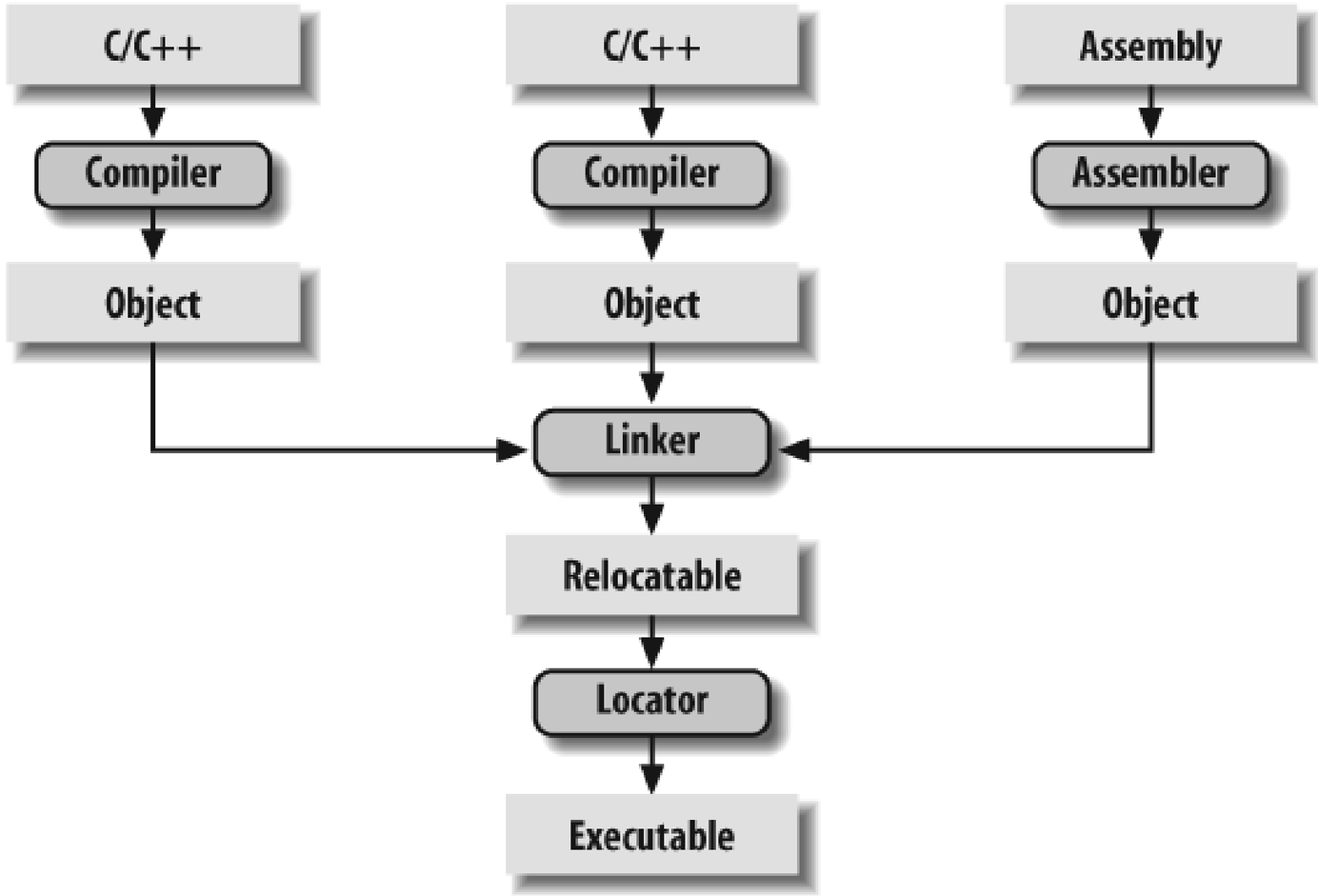
The embedded software development process just described is illustrated in figure,

the three steps are shown from top to bottom, with the tools that perform

the steps shown in boxes that have rounded corners.

1. Each of these development tools takes one or more files as input and produces a single output file.
2. More specific information about these tools and the files they produce is provided in the sections that follow.

C++ PROGRAM COMPILERS

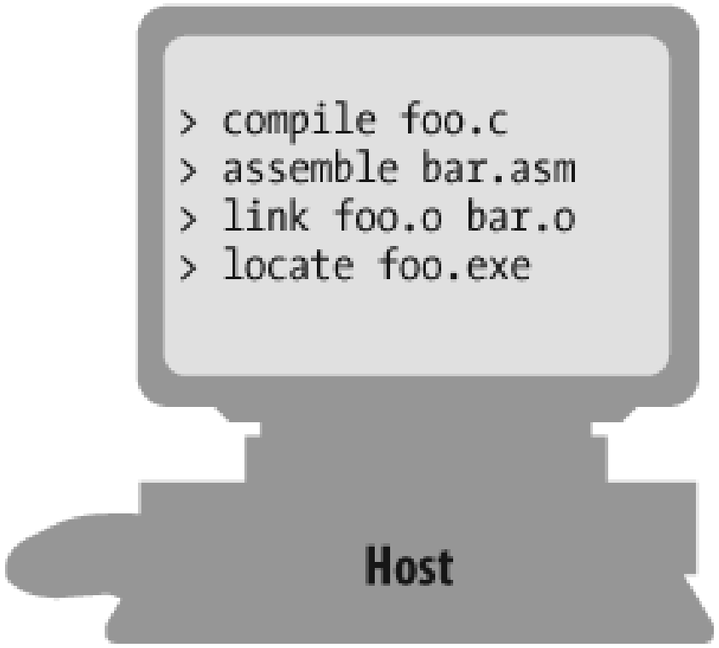


The embedded software development process

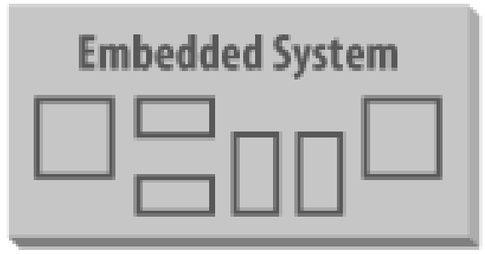
C++ PROGRAM COMPILERS

1. Each of the steps of the embedded software build process is a transformation performed by software running on a general-purpose computer.
2. To distinguish this development computer (usually a PC or Unix workstation) from the target embedded system, it is referred to as the host computer.
3. The **compiler**, assembler, linker, and locator run on a host computer rather than on the embedded system itself.
4. Yet, these tools combine their efforts to produce an executable binary image that will execute properly only on the target embedded system

C++ PROGRAM COMPILERS



The development tools that build the embedded software run on a general-purpose computer



The embedded software that is built by those tools runs on the embedded system

The split between host and target

C++ PROGRAM COMPILERS

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.

C++ PROGRAM cross COMPILERS

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.

C++ PROGRAM COMPILERS

The GNU C compiler (*gcc*) and assembler (*as*) can be configured as either native compilers or cross-compilers.

These tools support an impressive set of host-target combinations. The *gcc* compiler will run on all common PC and Mac operating systems.

The target processor support is extensive, including AVR, Intel x86, MIPS, PowerPC, ARM, and SPARC.

Additional information about *gcc* can be found online at http://gcc.gnu.org.

C++ PROGRAM COMPILERS

Regardless of the input language (C, C++, assembly, or any other), the output of the cross-compiler will be an object file.

This is a specially formatted binary file that contains the set of instructions and data resulting from the language translation process.

Although parts of this file contain executable code, the object file cannot be executed directly.

In fact, the internal structure of an object file emphasizes the incompleteness of the larger program.

C++ PROGRAM COMPILERS

The contents of an object file can be thought of as a very large, flexible data structure.

The structure of the file is often defined by a standard format such as the Common Object File Format (COFF) or Executable and Linkable Format (ELF).

If you'll be using more than one compiler (i.e., you'll be writing parts of your program in different source languages), you need to make sure that each compiler is capable of producing object files in the same format; *gcc* supports both of the file formats previously mentioned.

Although many compilers (particularly those that run on Unix platforms) support standard object file formats such as COFF and ELF, some others produce object files only in proprietary formats.

If you're using one of the compilers in the latter group, you might find that you need to get all of your other development tools from the same vendor.

C++ PROGRAM COMPILERS

•Most object files begin with a header that describes the sections that follow.

•Each of these sections contains one or more blocks of code or data that originated within the source file you created.

•However, the compiler has regrouped these blocks into related sections. For example, in *gcc* all of the code blocks are collected into a section called text, initialized global variables (and their initial values) into a section called data, and uninitialized global variables into a section called bss.

C++ PROGRAM COMPILERS

•There is also usually a symbol table somewhere in the object file that contains the names and locations of all the variables and functions referenced within the source file.

•Parts of this table may be incomplete, however, because not all of the variables and functions are always defined in the same file.

•These are the symbols that refer to variables and functions defined in other source files.

•And it is up to the linker to resolve such unresolved references.

C++ PROGRAM COMPILERS

Compile

As we have implemented it, the Blinking LED example consists of two source modules:

1. *led.c*
2. *blink.c*.

The first step in the build process is to compile these two files.

The basic structure for the *gcc* compiler command is:

arm-elf-gcc [

options

]

file **...**

C++ PROGRAM COMPILERS

The command-line options we'll need are:

-g

To generate debugging info in default format

-c

To compile and assemble but not link

-Wall

To enable most warning messages

-I../include

To look in the directory *include* for header files

Here are the actual commands for compiling the C source files:

 # arm-elf-gcc –g -c –Wall -I../include led.c

 # arm-elf-gcc -g –c -Wall -I../include blink.c

C++ PROGRAM COMPILERS

We broke up the compilation step into two separate commands, but you can compile the two files with one command.

To use a single command, just put both of the source files after the options.

If you wanted different options for one of the source files, you would need to compile it separately as just shown.

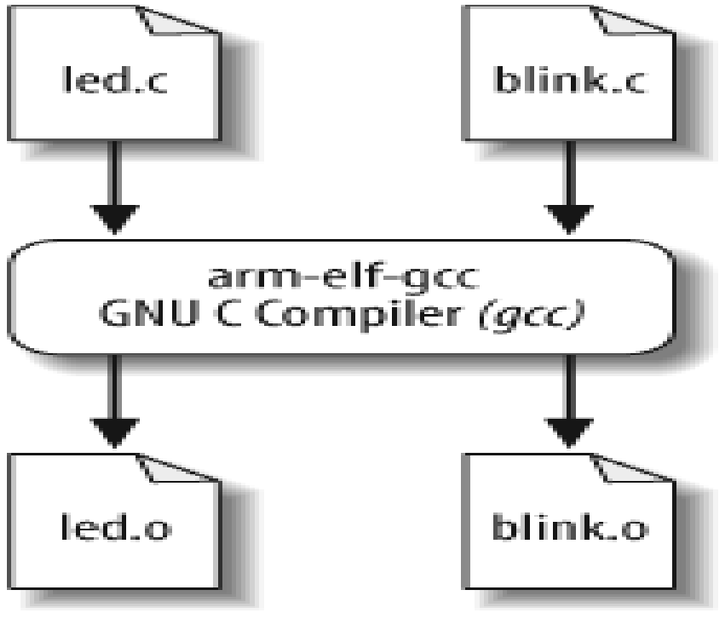
For additional information about compiler options, take a look at http://gcc.gnu.org.

C++ PROGRAM COMPILERS

Running these commands will be a good way to verify that the tools were set up properly.

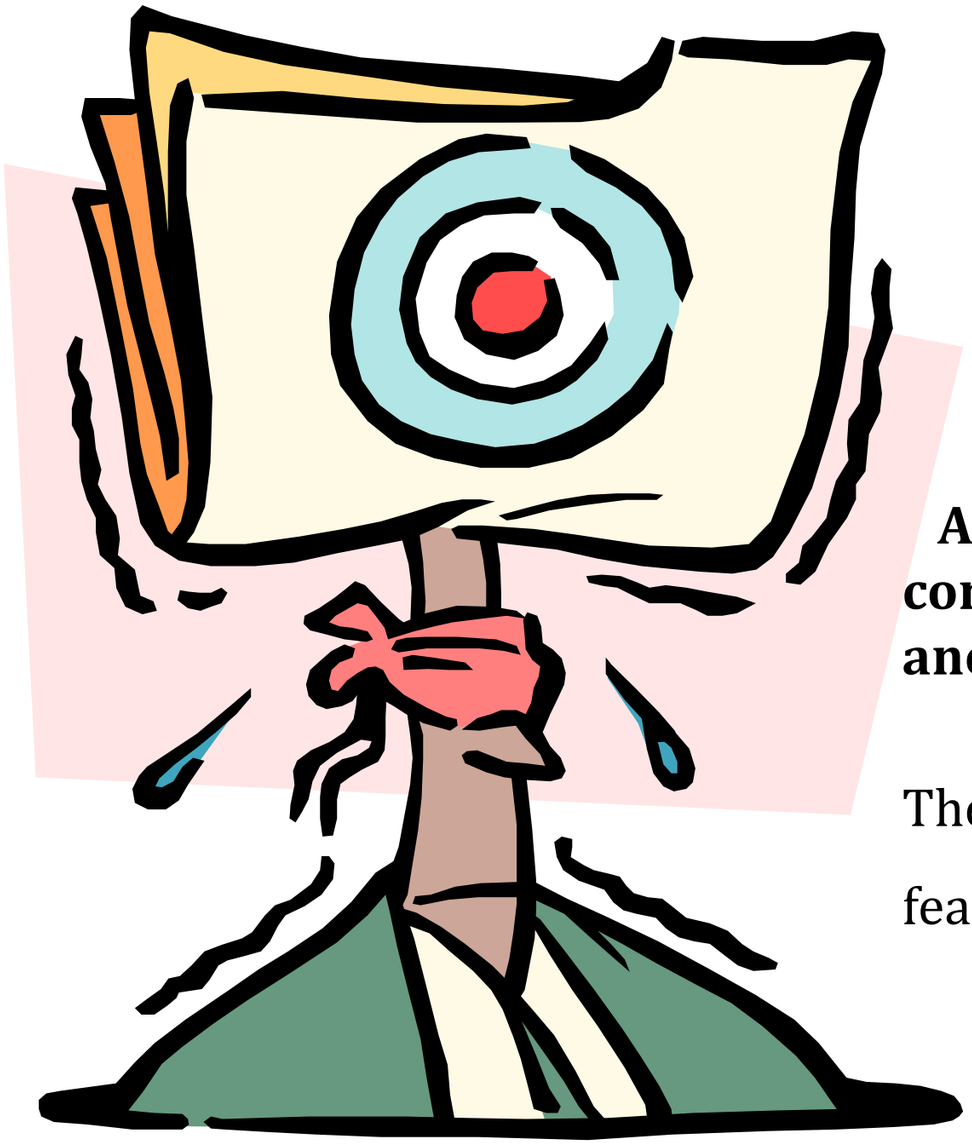
The result of each of these commands is the creation of an object file that has the same prefix as the *.c* file, and the extension *.o*.

So if all goes well, there will now be two additional files—*led.o* and *blink.o*—in the working directory. The compilation procedure is shown in Figure



Compiling the Blinking LED program

Any Questions / Thank you



Shoot!

A compiler such as this—that runs on one computer platform and produces code for another—is called a *cross-compiler*.

(Next Class)

The use of a cross-compiler is one of the defining features of embedded software development.