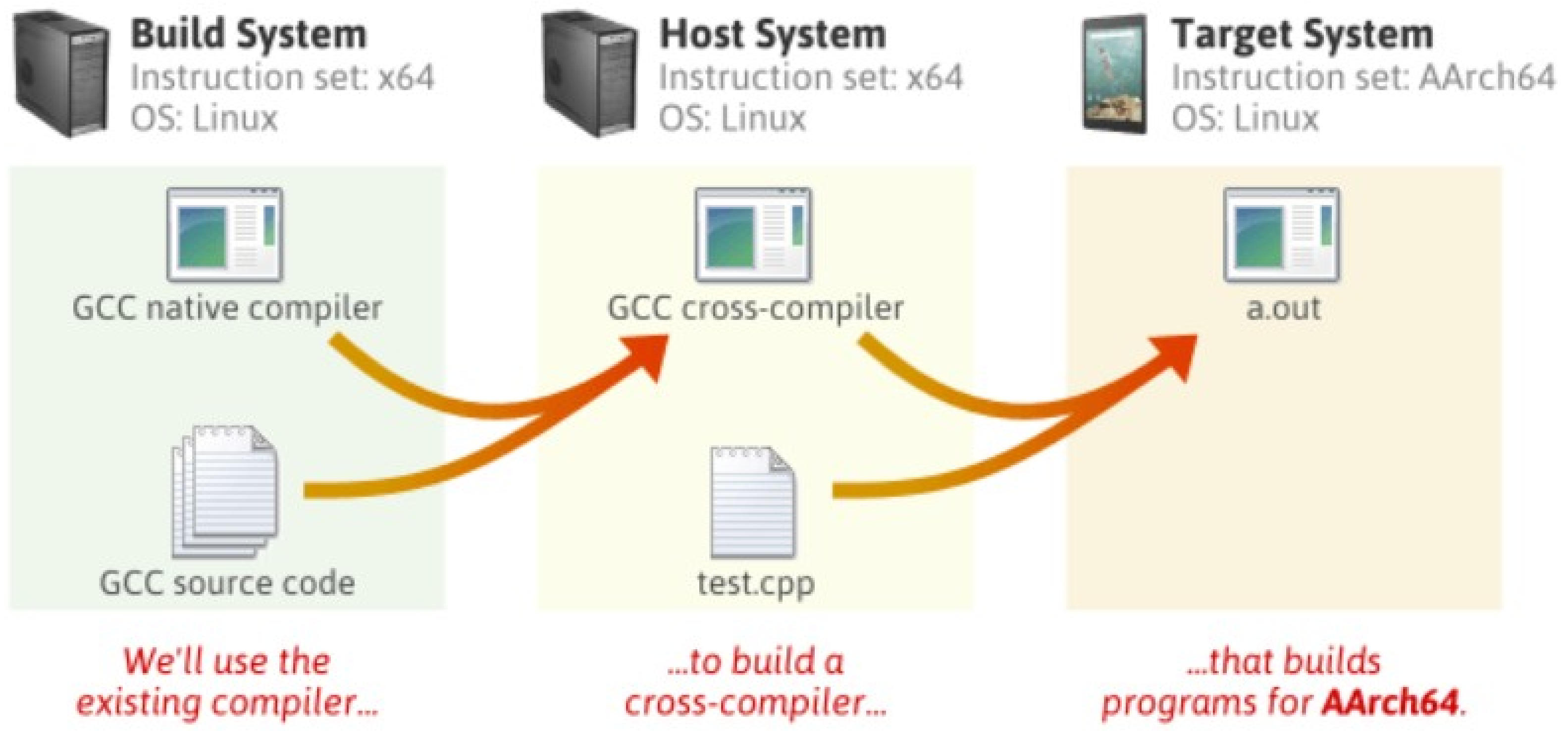
CH4 Development Tool

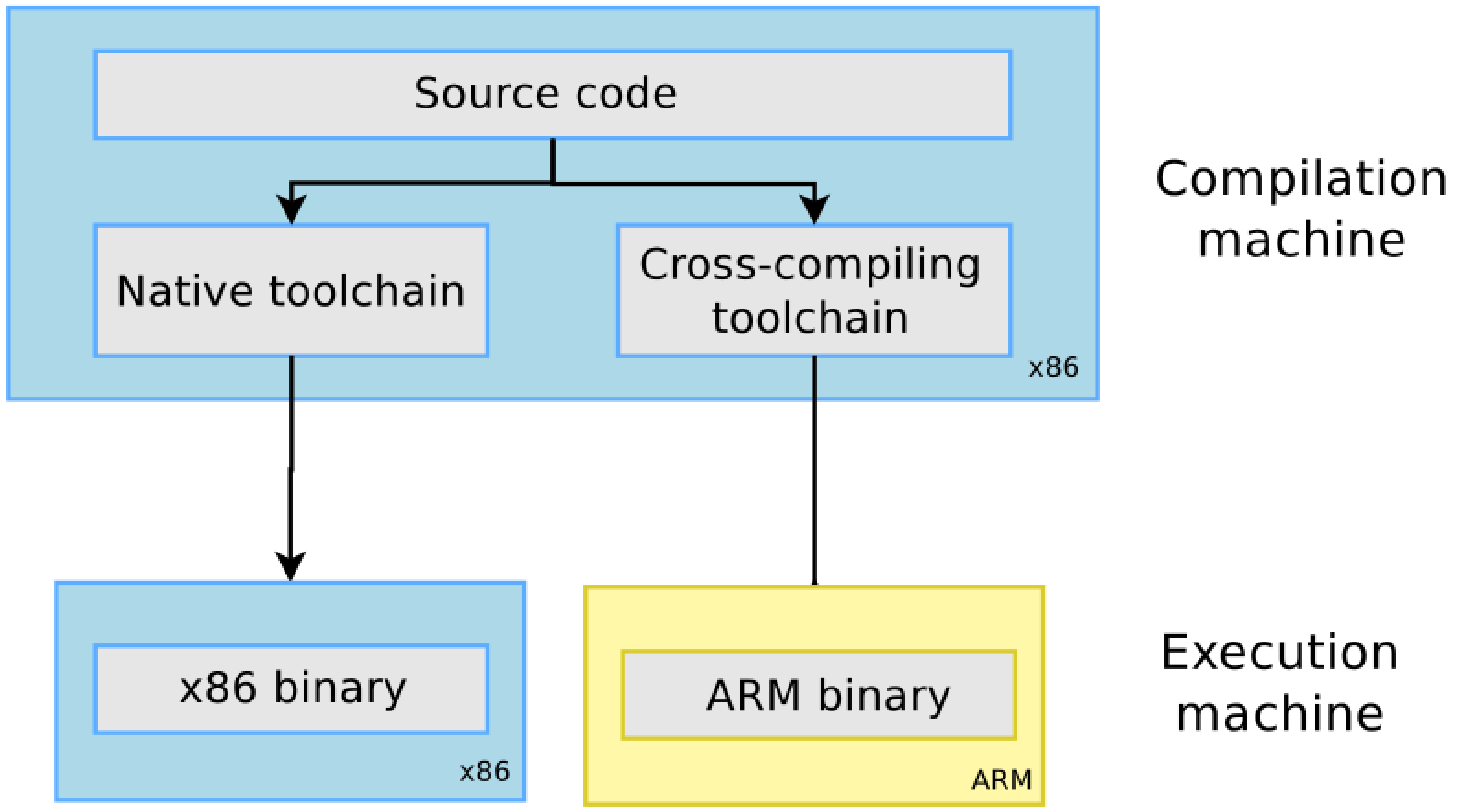
Cross Compilation Toolchain



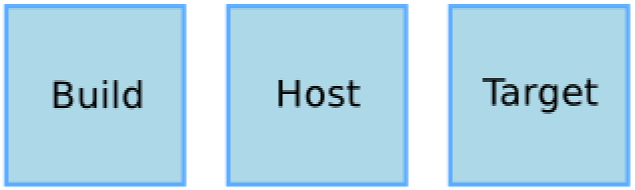
Cross Compilation Toolchain

- The Build machine, where the toolchain is built
- The Host machine, where the toolchain will be executed
- The Target machine, where the binaries created by the toolchain are executed.
-

Cross Compilation toolchain

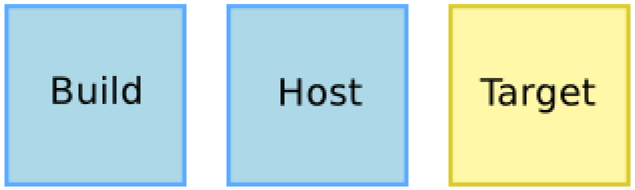


Cross Compilation Toolchain



Native build

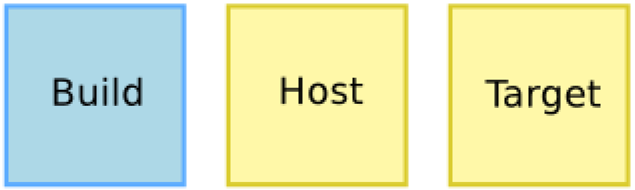
used to build the normal gcc
of a workstation



Cross build

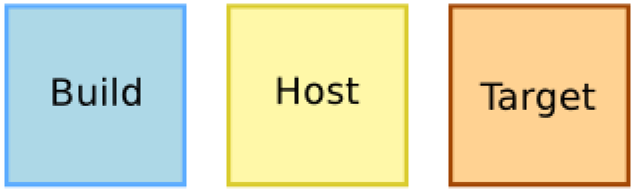
used to build a toolchain that runs
on your workstation but generates
binaries for the target

The most common case in embedded development



Cross-native build

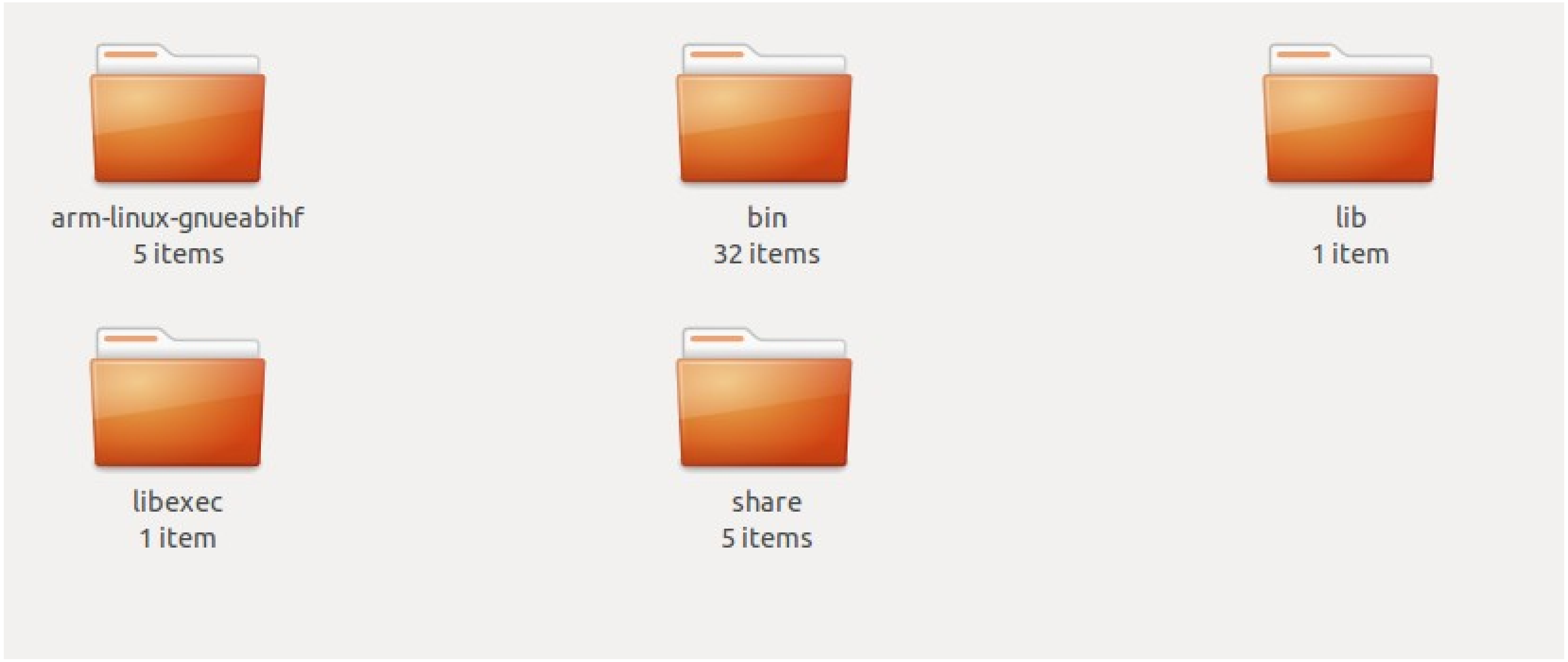
used to build a toolchain that runs on your
target and generates binaries for the target



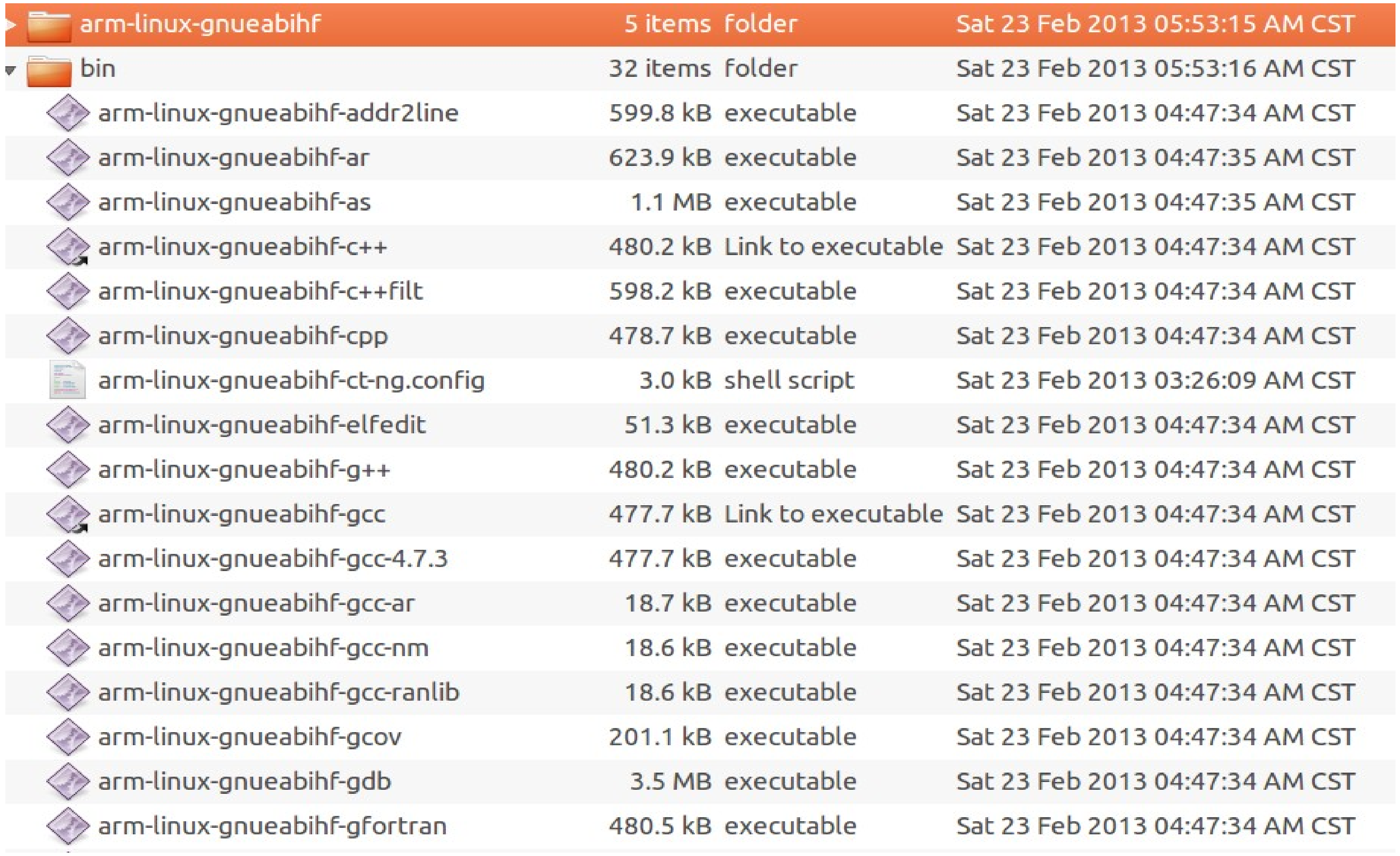
Canadian build

used to build on architecture A a
toolchain that runs on architecture B
and generates binaries for architecture C

Cross Compilation Toolchain

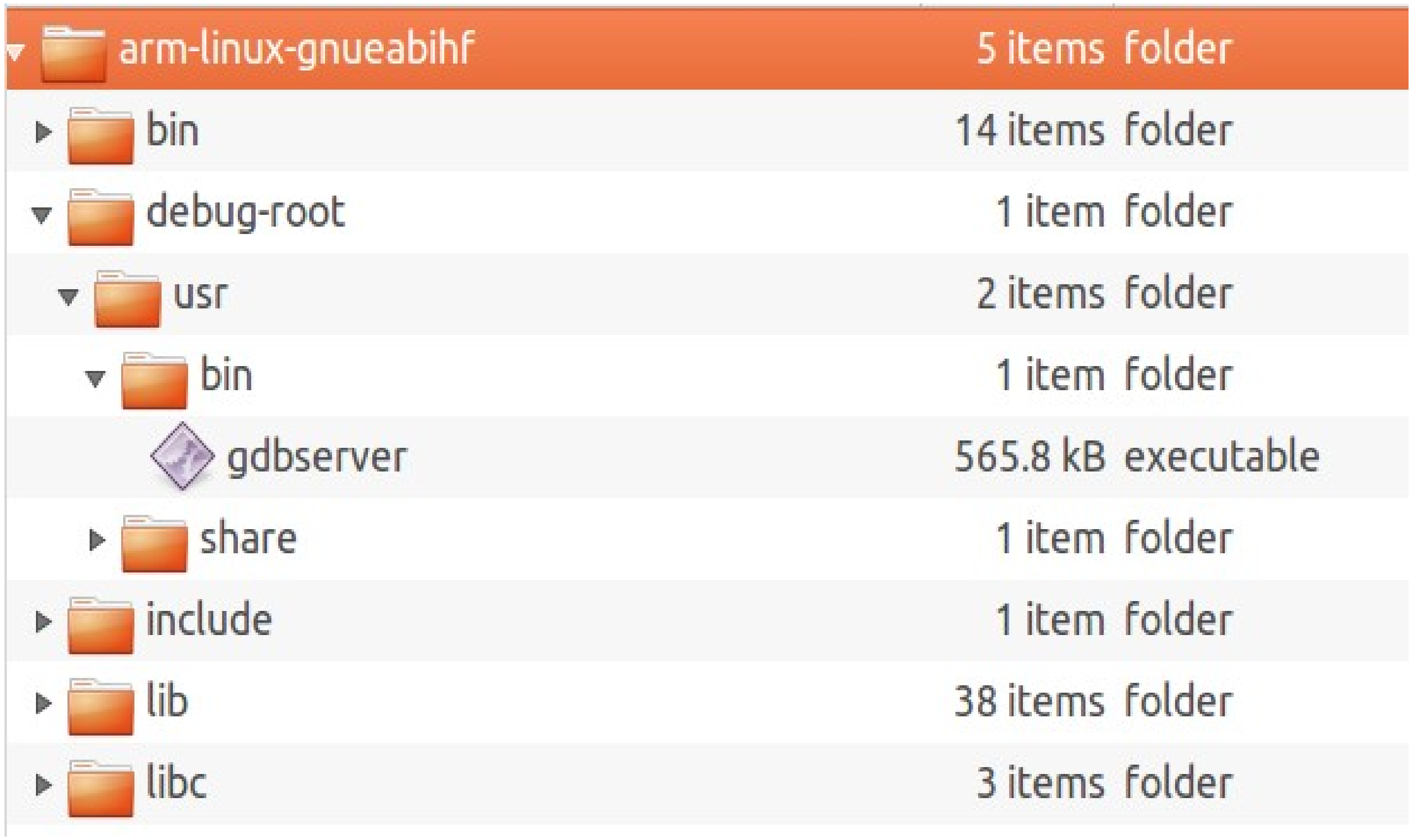


Cross Compilation Toolchain



arm-linux-gnueabihf	5 items	folder	Sat 23 Feb 2013 05:53:15 AM CST
bin	32 items	folder	Sat 23 Feb 2013 05:53:16 AM CST
arm-linux-gnueabihf-addr2line	599.8 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-ar	623.9 kB	executable	Sat 23 Feb 2013 04:47:35 AM CST
arm-linux-gnueabihf-as	1.1 MB	executable	Sat 23 Feb 2013 04:47:35 AM CST
arm-linux-gnueabihf-c++	480.2 kB	Link to executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-c++filt	598.2 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-cpp	478.7 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-ct-ng.config	3.0 kB	shell script	Sat 23 Feb 2013 03:26:09 AM CST
arm-linux-gnueabihf-elfedit	51.3 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-g++	480.2 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcc	477.7 kB	Link to executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcc-4.7.3	477.7 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcc-ar	18.7 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcc-nm	18.6 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcc-ranlib	18.6 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gcov	201.1 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gdb	3.5 MB	executable	Sat 23 Feb 2013 04:47:34 AM CST
arm-linux-gnueabihf-gfortran	480.5 kB	executable	Sat 23 Feb 2013 04:47:34 AM CST

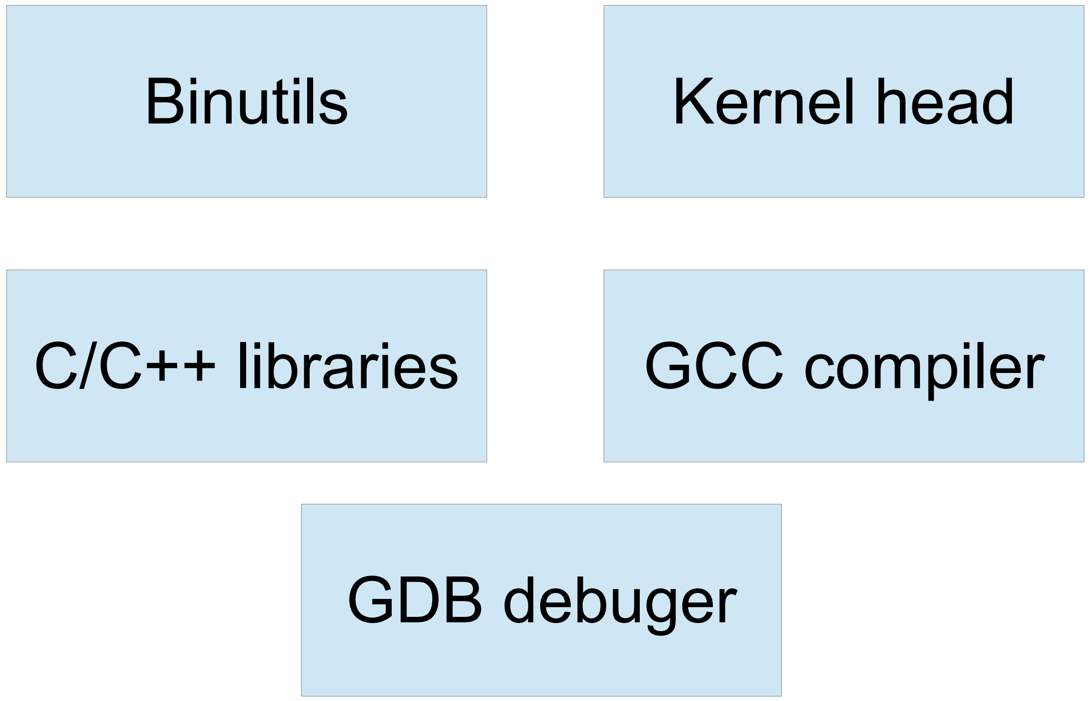
Cross Compilation Toolchain



arm-linux-gnueabihf	5 items	folder
bin	14 items	folder
debug-root	1 item	folder
usr	2 items	folder
bin	1 item	folder
gdbserver	565.8 kB	executable
share	1 item	folder
include	1 item	folder
lib	38 items	folder
libc	3 items	folder

GCC Components

- The GNU C Compiler
- The GNU Compiler Collection



Binutils

- Binutils
  - **as** : the assembler, that generates binary code from assembler source code
  - **ld** : the linker
  - **ar, ranlib** : to generate .a archives, used for libraries
  - **objdump, readelf, size, nm, strings** : to inspect binaries
  - **strip** : to strip useless parts of binaries in order to reduce their size

Kernel head

- The C library and compiled programs needs to interact with the kernel
- Compiling the C library requires kernel headers, and many applications also require them
- The kernel to user space ABI is backward compatible

ABI

- **ABI**, for **A**pplication **B**inary **I**nterface, defines the calling conventions
  - how function arguments are passed
  - how the return value is passed
  - how system calls are made
  - the organization of structures (alignment, etc.)
- All binaries in a system must be compiled with the same ABI, and the kernel must understand this ABI
- ARM - OABI and EABI

GCC

- GCC originally stood for the "GNU C Compiler."
- GNU Compiler Collection
    - C, C++, Ada, Objective-C, Fortran, JAVA ...
- http://gcc.gnu.org/

GCC flag

- arm-linux-gnueabihf-gcc –help
- -c : Compile and assemble, but do not link
- -o <file> : Place the output into <file>
- -shared : Create a shared library
- -g : add debug information
- -O : sets the compiler's optimization level
- -Wall : enables all compiler's warning messages
- -D : defines a macro to be used by the preprocessor
- -I : adds include directory of header files
- -L,-l :
  - -L looks in directory for library files
  - -l links with a library file

C library

- The C library is an essential component of a Linux system
- Several C libraries are available:
  - **glibc, uClibc, eglibc, dietlibc, newlib**
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.

C library

- Executable size comparison on ARM, tested with eglibc 2.15 and uClibc 0.9.33.2

Plain “hello world” program (stripped):

helloworld	static	dynamic
uClibc	18kB	2.5kB
uClibc with Thumb-2	14kB	2.4kB
eglibc with Thumb-2	361kB	2.7kB

Busybox (stripped):

busybox	static	dynamic
uClibc	750kB	603kB
uClibc with Thumb-2	533kB	439kB
eglibc with Thumb-2	934kB	444kB

Floating point support

- For processors having a **floating point unit**, the toolchain should generate hard float code, in order to use the floating point instructions directly
- For processors without a floating point unit
  - Generate hard float code and rely on the kernel to emulate the floating point instructions
  - Generate soft float code, so that instead of generating floating point instructions, calls to a user space library are generated

Floating point support

- GCC floating-point options
  - -mfloat-abi = below
    - **soft**: Full software floating point
    - **softfp**: Use the FPU, but remain compatible with soft-float code
    - **hard**: Full hardware floating point

CPU optimization flags

- A set of cross-compiling tools is specific to a CPU architecture
- We use the **-march=**, **-mcpu=**, **-mtune**= options, one can select more precisely the target CPU type
  - For example : -march=armv7 -mcpu=cortex-a8
- ARM Compiler armclang Reference Guide

Obtain a Toolchain

- Building a cross-compiling toolchain by ourself
  - Crosstool-NG
  - http://crosstool-ng.org/#introduction
- Pre-build toolchain
  - Linaro - https://wiki.linaro.org/WorkingGroups/ToolChain
  - By Linux distribution -
    - sudo apt-get install gcc-arm-linux-gnueabi
  - CodeSourcery
  - BSP

Installing and using a pre-compiled toolchain

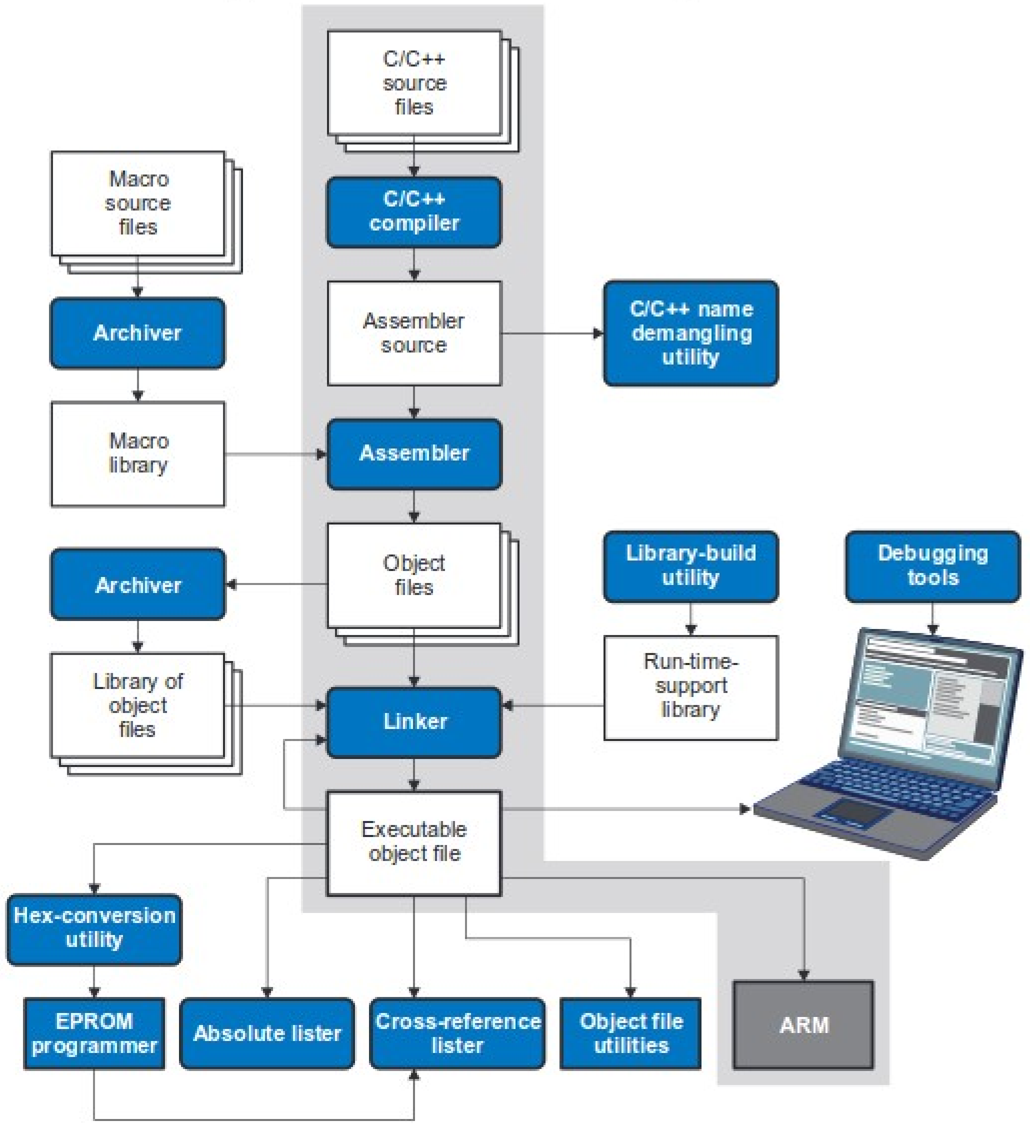
- Add the path to toolchain binaries in your PATH: export
  - PATH=/path/to/toolchain/bin/:$PATH
- Compile your applications
  - PREFIX-gcc -o testme testme.c
- PREFIX
  - depends on the toolchain configuration

Toolchain building utilities

- **Buildroot**
  - Makefile-based
  - http://www.buildroot.net
- **PTXdist**
  - Makefile-based
  - http://pengutronix.de/software/ptxdist/
- **OpenEmbedded / Yocto**
  - A featureful, but more complicated build system
  - http://www.openembedded.org/
  - https://www.yoctoproject.org/

Compile, Assembler, Linker

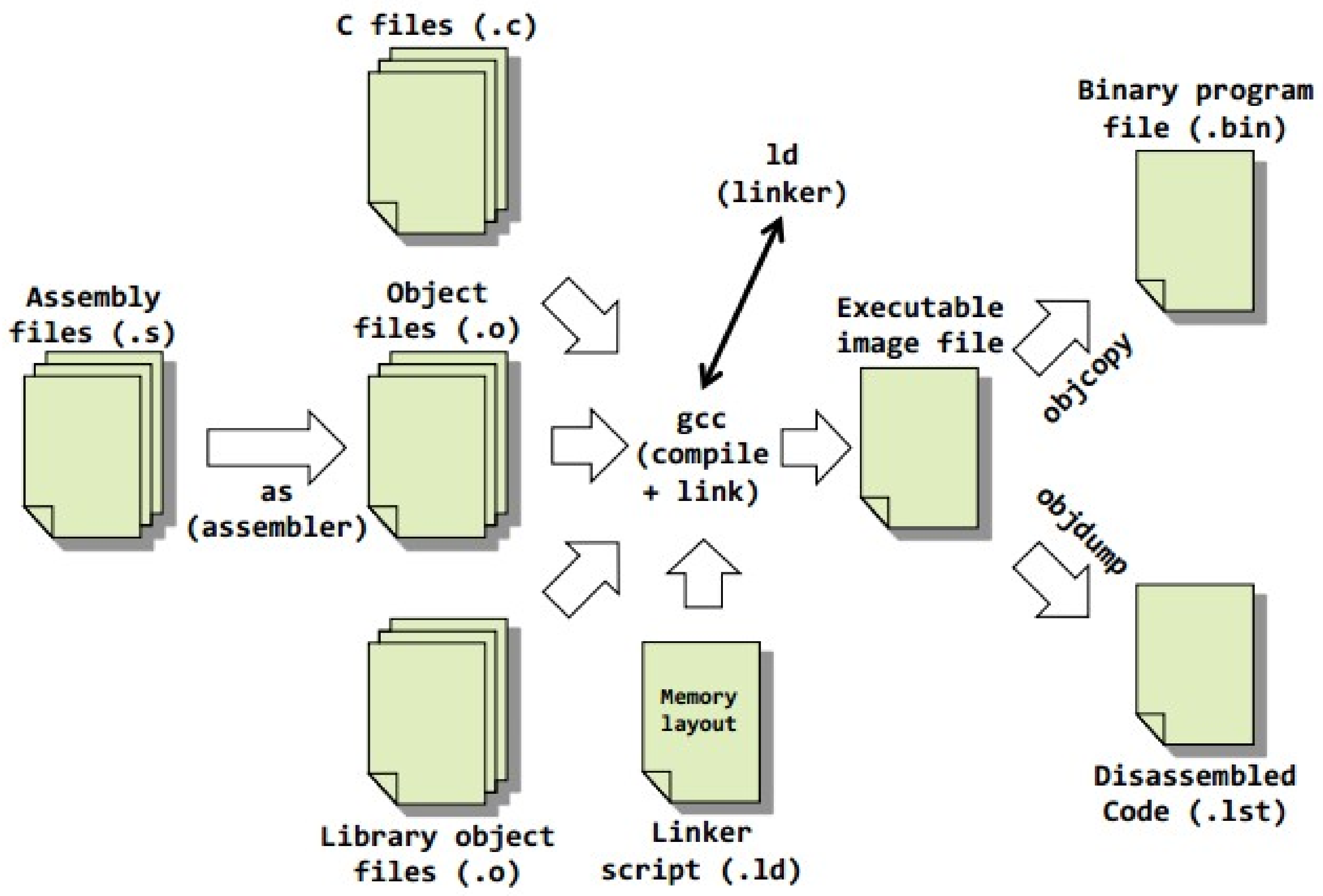
Software Development Tools Overview



Tools Descriptions

- C/C++ compiler
  - produces ARM machine code object modules
- Assembler
  - Translates Assembly Language Source Files Into Machine Language Object modules
- Linker
  - Combines object files into a single executable object module

Executable Object Files



OBJECT FILES and EXECUTABLE

Object File Format	Description
a.out	The a.out format is the original file format for Unix. It consists of three sections: text, data, and bss, which are for program code, initialized data, and uninitialized data, respectively. This format is so simple that it doesn't have any reserved place for debugging information. The only debugging format for a.out is stabs, which is encoded as a set of normal symbols with distinctive attributes.
COFF	The COFF (Common Object File Format) format was introduced with System V Release 3 (SVR3) Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging information was limited. There is no file extension for this format.
ECOFF	A variant of COFF. ECOFF is an Extended COFF originally introduced for Mips and Alpha workstations.
XCOFF	The IBM RS/6000 running AIX uses an object file format called XCOFF (e**X**tended COFF). The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the .debug section (rather than the string table). The default name for an XCOFF executable file is a.out.
PE	Windows 9x and NT use the PE (Portable Executable) format for their executables. PE is basically COFF with additional headers. The extension normally .exe.
ELF	The ELF (Executable and Linking Format) format came with System V Release 4 (SVR4) Unix. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF used on most modern Unix systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems.
SOM/ESOM	SOM (System Object Module) and ESOM (Extended SOM) is HP's object file and debug format (not to be confused with IBM's SOM, which is a cross-language Application Binary Interface - ABI).

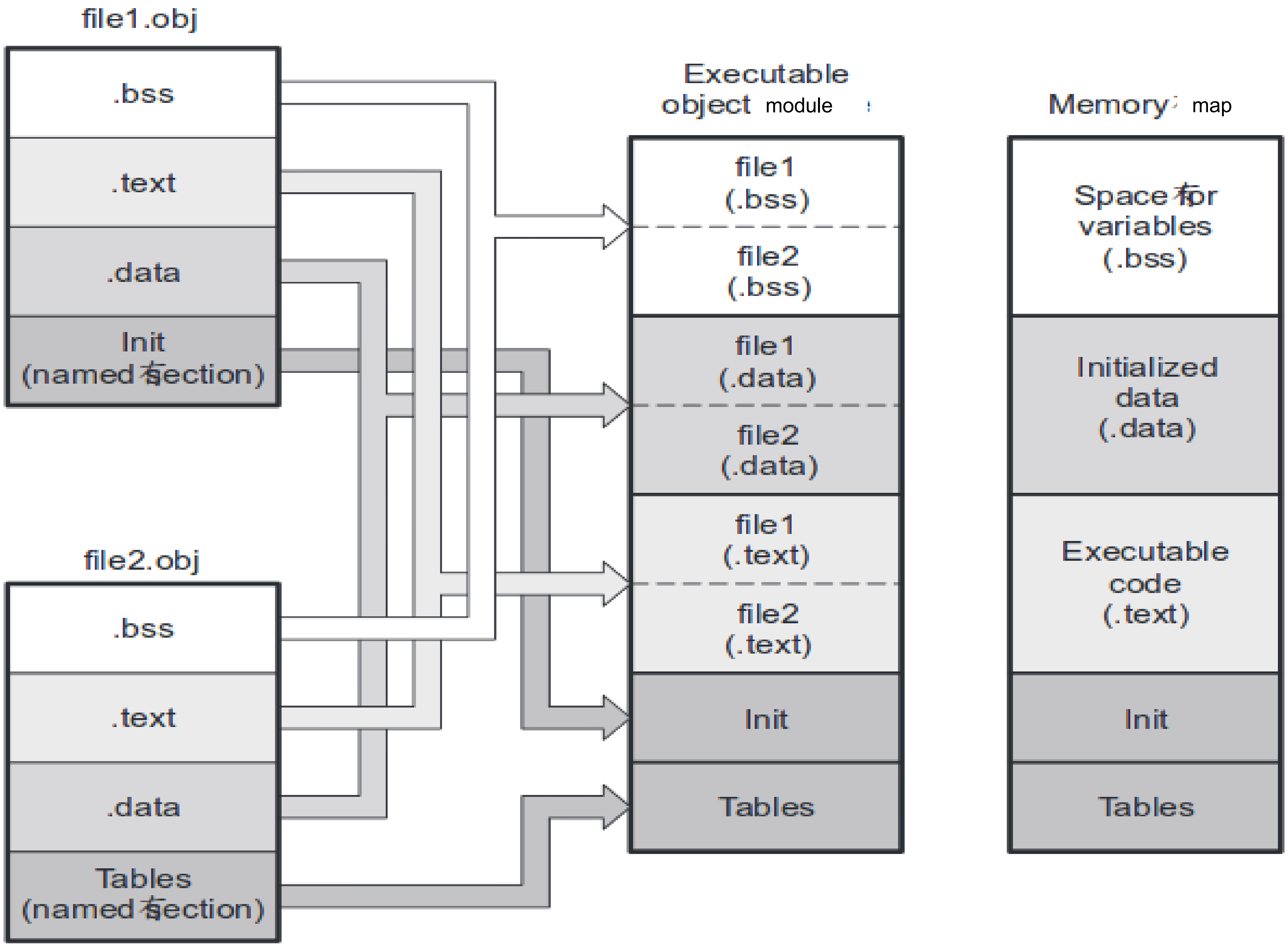
Introduction to Sections

- .text section
  - contains executable code
- .data section
  - usually contains initialized data
- .bss section
  - uninitialized variables

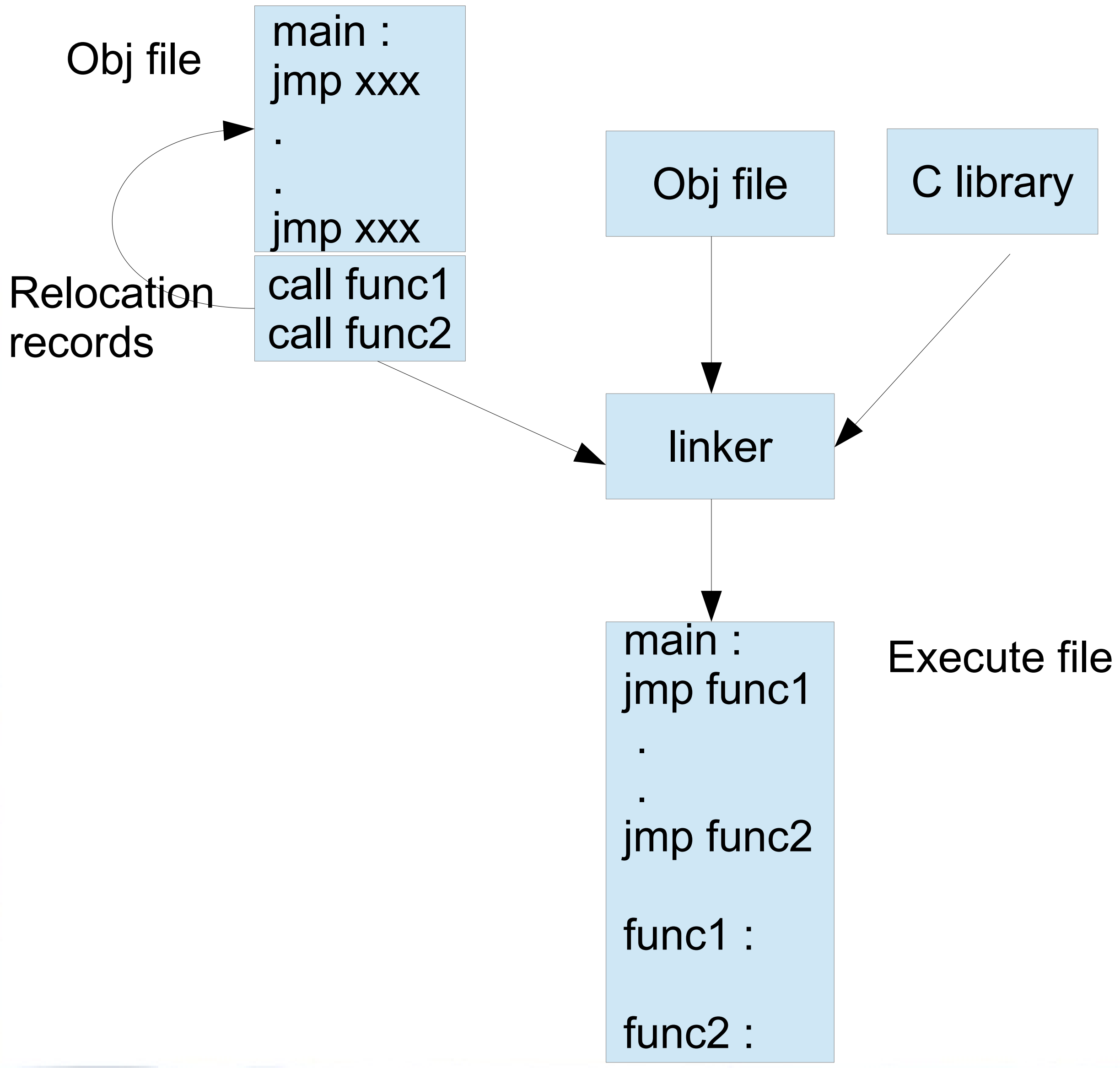
Introductio to Sections

Section	Description
.text	This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.bss	BSS stands for 'Block Started by Symbol'. It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Also known as .rodata (read-only data) section. This contains constants and string literals.
.reloc	Stores the information required for relocating the image while loading.
Symbol table	A symbol is basically a name and an address. Symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The symbol table contains an array of symbol entries.
Relocation records	Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Re-locatable files must have relocation entries' which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Simply said relocation records are information used by the linker to adjust section contents.

Combine Input Sections



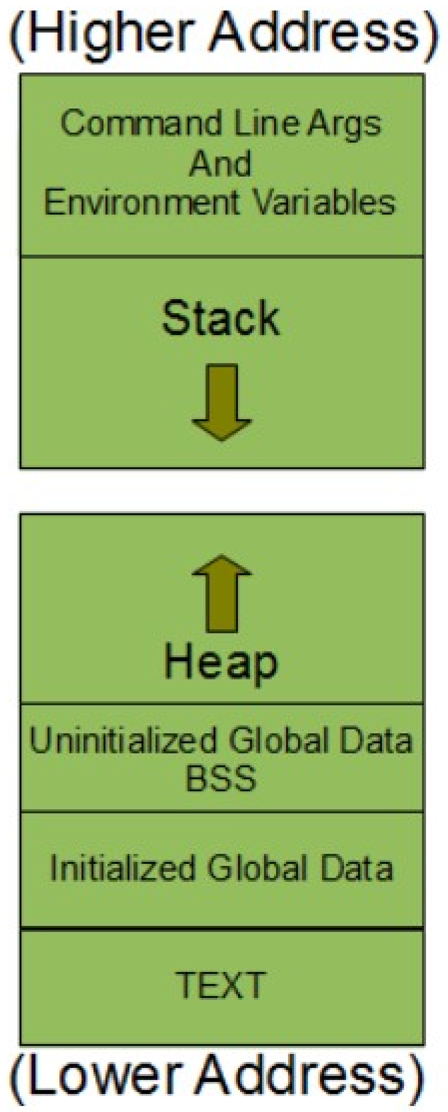
The relocation record



Load and Run Addresses

- Variable data in the program must be **writable**, and so must be located in **writable memory**, typically RAM
- The **load address** is the location of an object in the load image
- The **run address** is the location of the object as it exists during program execution
- The **load and run addresses** for an object may be the same
  - Read-only data
- The **load and run addresses** for an object may be the different
  - Writable data

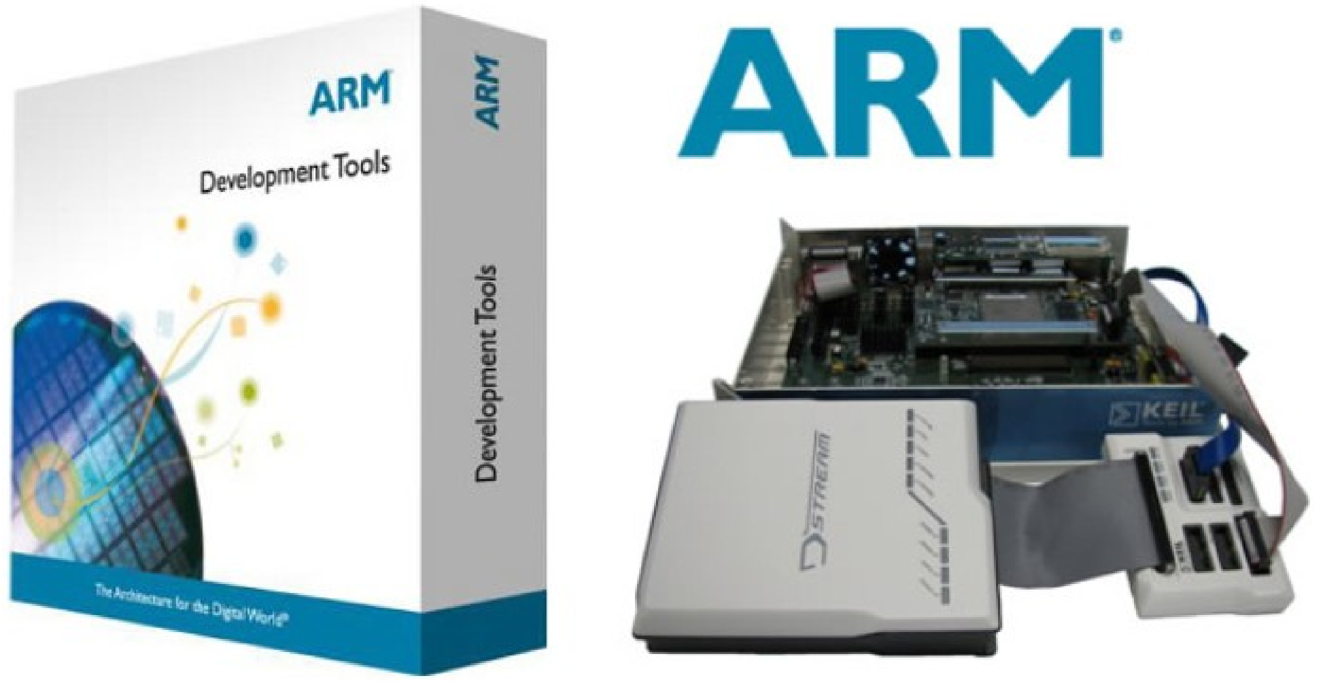
Memory Layout of a Process


(Higher Address)
Command Line Args And Environment Variables
Stack
Heap
Uninitialized Global Data BSS
Initialized Global Data
TEXT
(Lower Address)

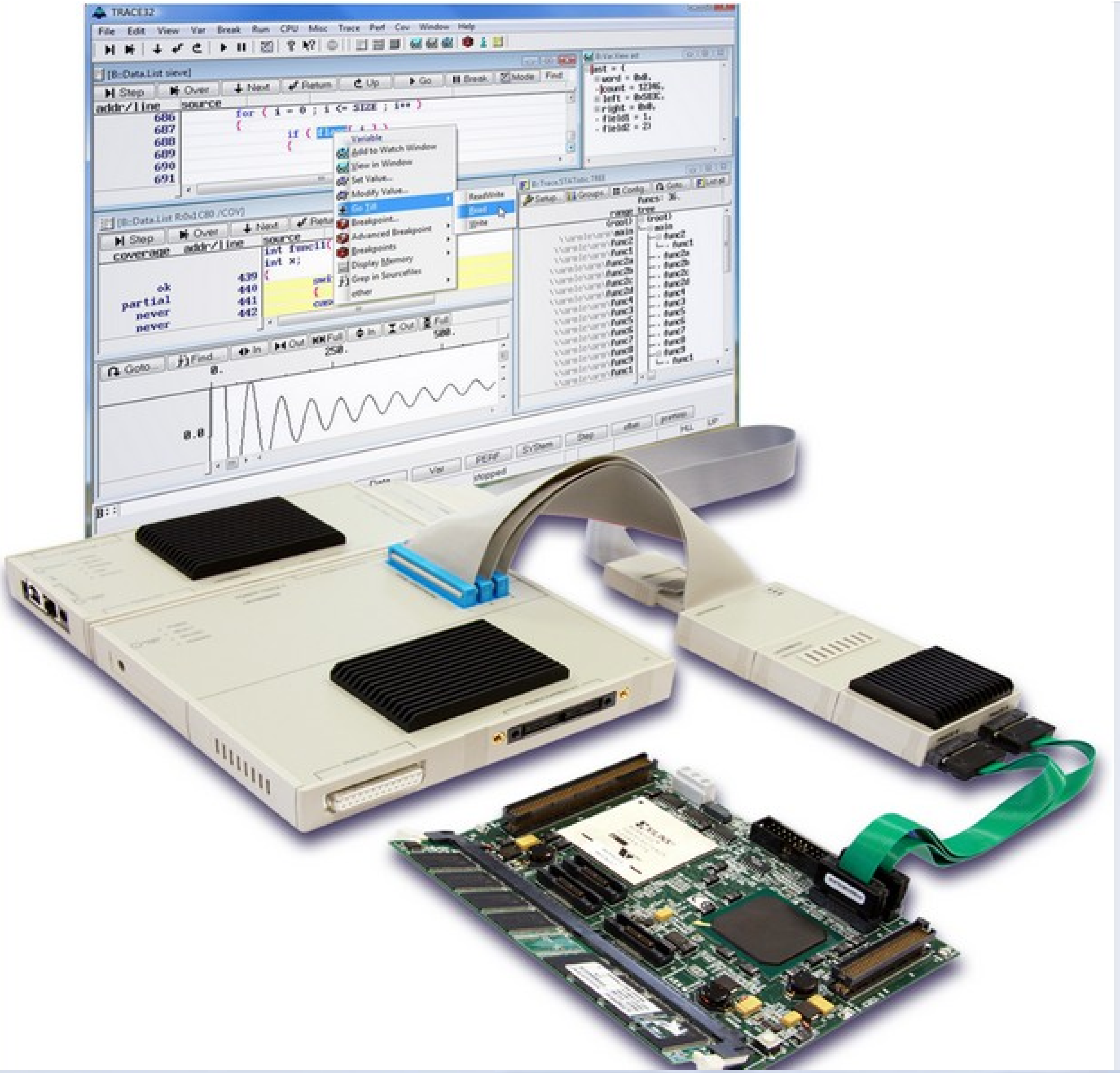
Debug Tool

- Hardware
  - DS-5
  - Trace32
- Software
  - GDB - The GNU Project Debugger

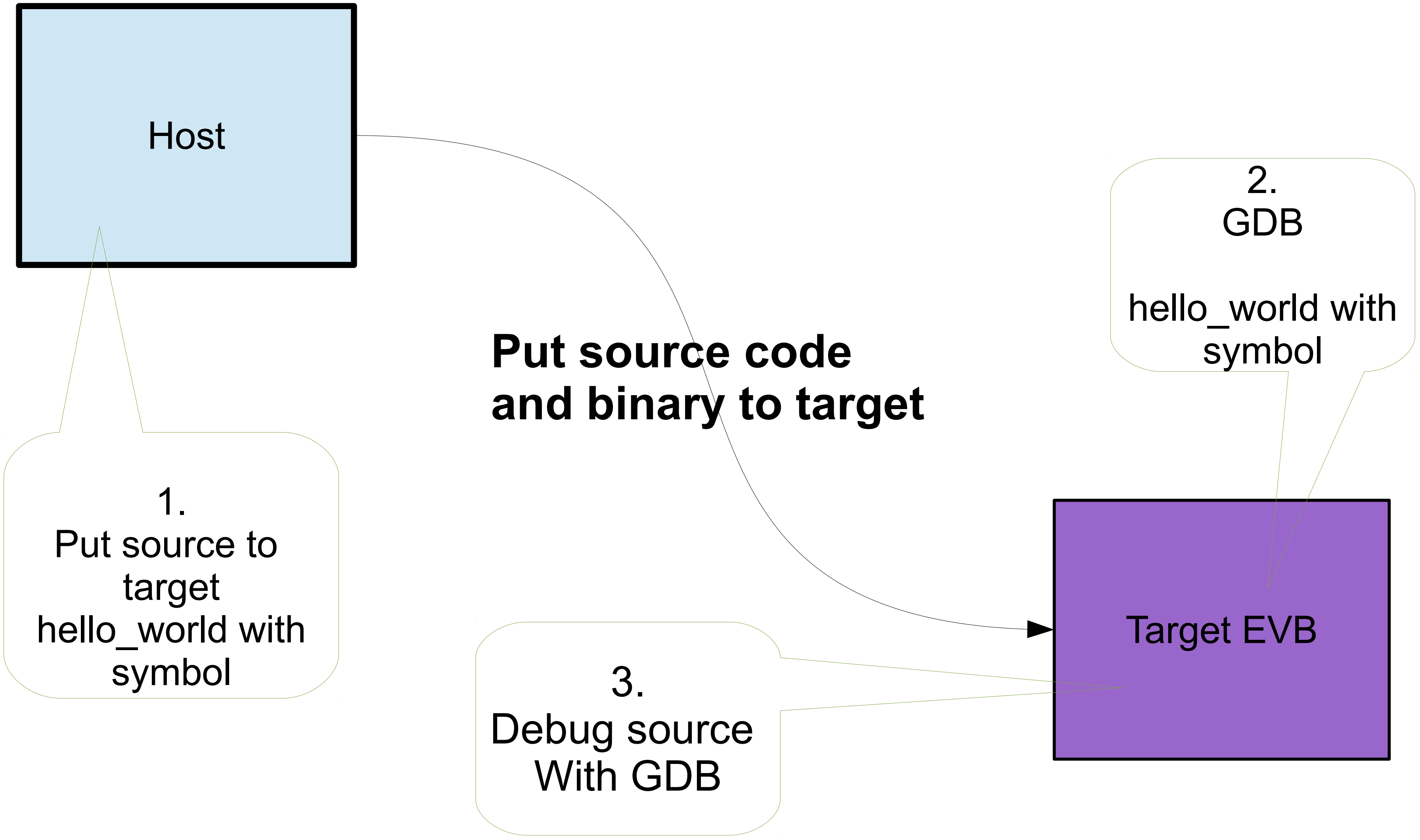
DS-5


ARM
Development Tools
The Architecture for the Digital World
ARM
DSTREAM
KEIL

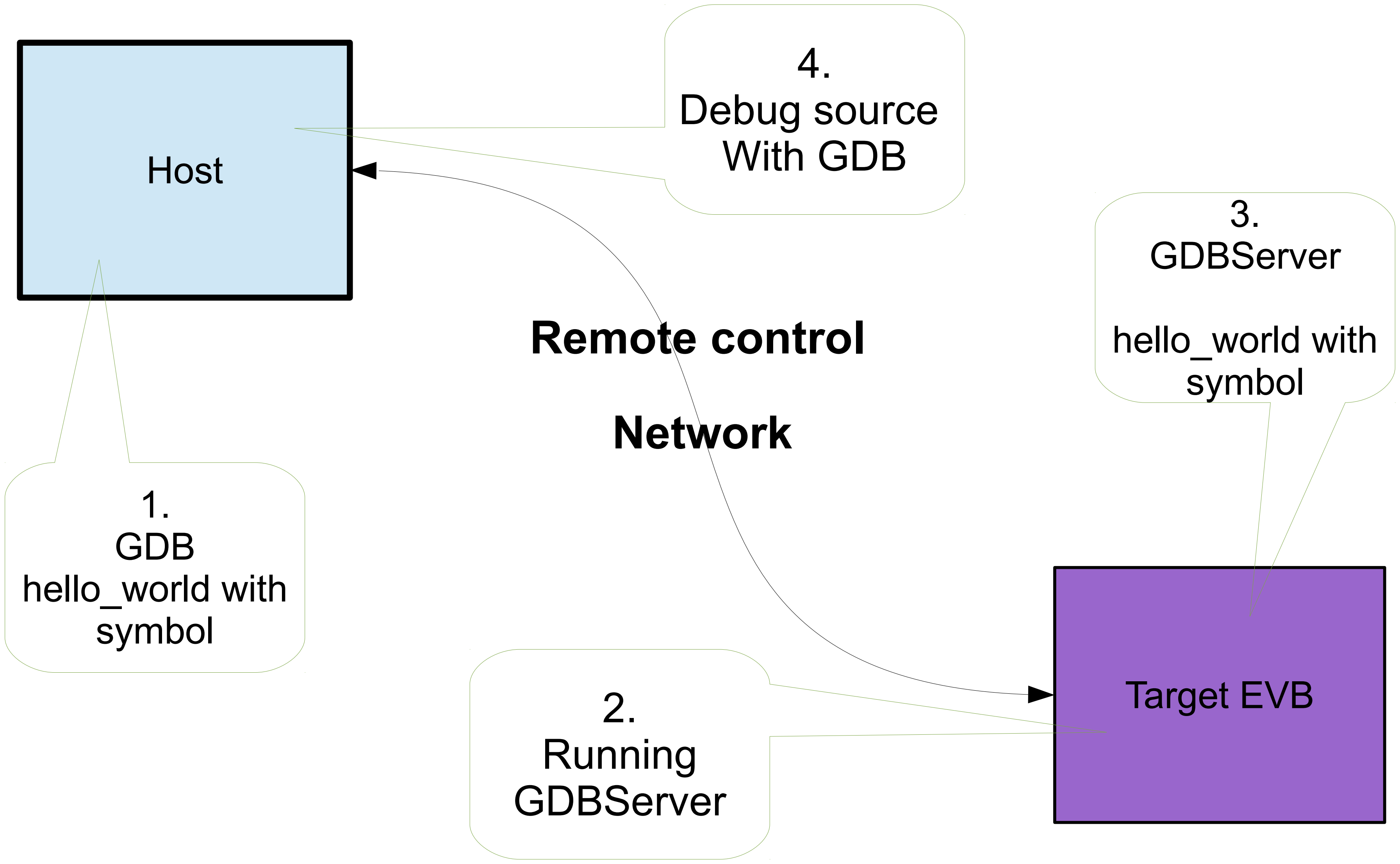
Trace32



GDB with Local


Host
Put source code and binary to target
2. GDB hello_world with symbol
1. Put source to target hello_world with symbol
Target EVB
3. Debug source With GDB

GDB with Remote



GDB

- The GNU Project Debugger : websit
- GDB can do four main kinds of things
  - Start your program, specifying anything that might affect its behavior
  - Make your program stop on specified conditions
  - Examine what has happened, when your program has stopped
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another

Build GDB Environment

- Local
  - GDB for target : arm-linux-gnueabihf-gdb
- Remote
  - GDB for Host : arm-linux-gnueabihf-gdb
  - GDBServer for target : gdbserver

GDB basic command

Start and Stop	
run r run *command-line-arguments* run < *infile* > *outfile*	Start program execution from the beginning of the program. The command `break main` will get you started. Also allows basic I/O redirection.
continue c	Continue execution to next break point.
kill	Stop program execution.
quit q	Exit GDB debugger.

GDB basic command

Command	Description
info breakpoints	List breakpoints
info break	List breakpoint numbers.
info break *breakpoint-number*	List info about specific breakpoint.
info watchpoints	List breakpoints
info registers	List registers in use
info threads	List threads in use
info set	List set-able option

GDB basic command

Source Code	
list l list *line-number* list *function* list - list *start#,end#* list *filename:function*	List source code.
set listsize *count* show listsize	Number of lines listed when `list command given.`
directory *directory-name* dir *directory-name* show directories	Add specified directory to front of source code path.
directory	Clear sourcepath when nothing specified.

GDB basic command

Stack	
backtrace bt bt *inner-function-nesting-depth* bt -*outer-function-nesting-depth*	Show trace of where you are currently. Which functions you are in. Prints stack backtrace.
backtrace full	Print values of local variables.
frame frame *number* f *number*	Show current stack frame (function where you are stopped) Select frame number. (can also user up/down to navigate frames)
up down up *number* down *number*	Move up a single frame (element in the call stack) Move down a single frame Move up/down the specified number of frames in the stack.
info frame	List address, language, address of arguments/local variables and which registers were saved in frame.
info args info locals info catch	Info arguments of selected frame, local variables and exception handlers.

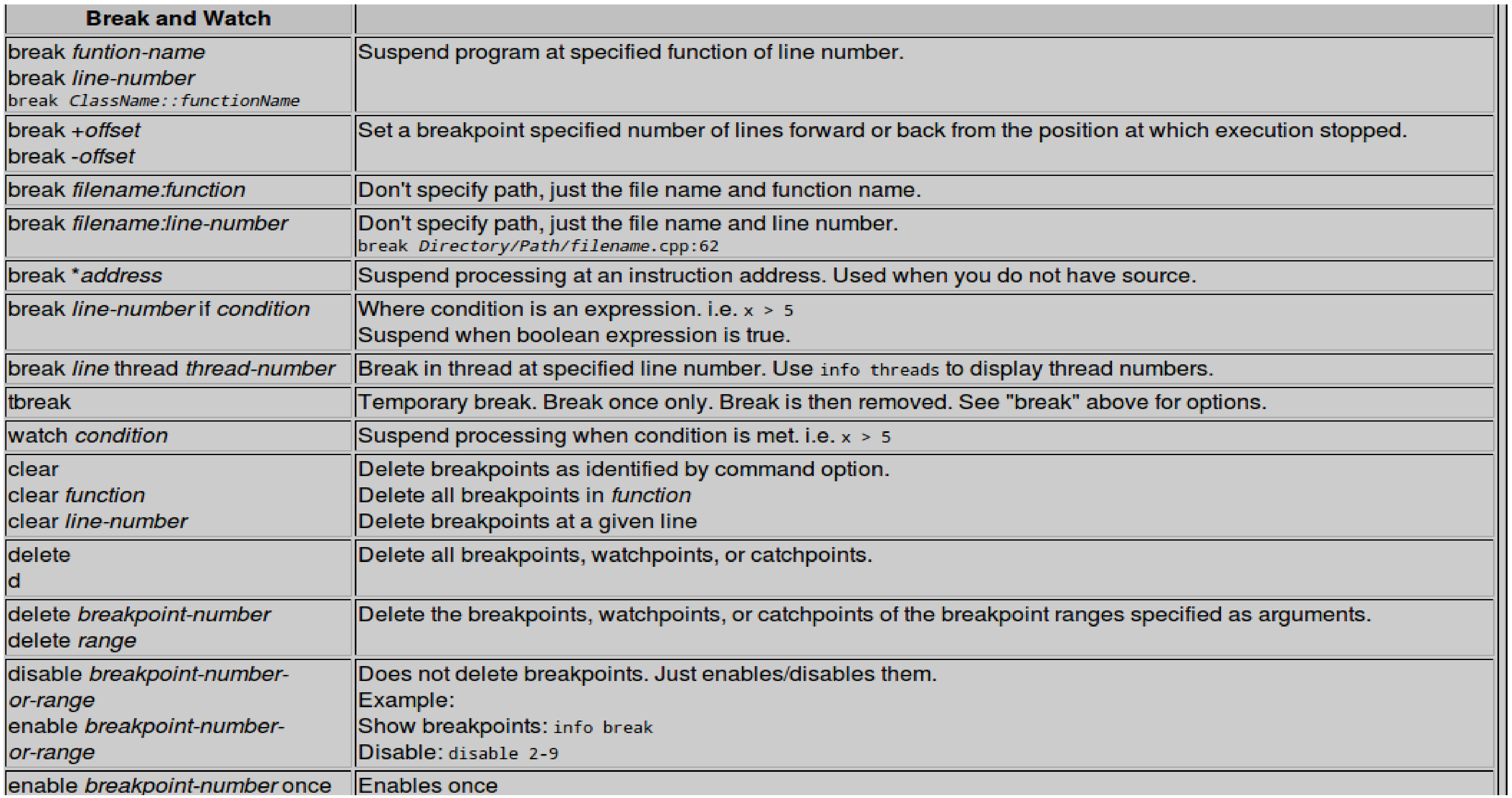
GDB basic command

Line Execution	
step s step *number-of-steps-to-perform*	Step to next line of code. Will step into a function.
next n next *number*	Execute next line of code. Will not enter functions.

until
until *line-number*
	Continue processing until you reach a specified line number. Also: function name, address, filename:function or filename:line-number.

info signals info handle handle *SIGNAL-NAME option*	Perform the following option when signal recieved: nostop, stop, print, noprint, pass/noignore or nopass/ignore
where	Shows current line number and which function you are in.

GDB basic command



Break and Watch	

break *funtion-name*
break *line-number*
`break ClassName::functionName`
	Suspend program at specified function of line number.

break +*offset*
break -*offset*
	Set a breakpoint specified number of lines forward or back from the position at which execution stopped.

break *filename:function*	Don't specify path, just the file name and function name.
break *filename:line-number*	Don't specify path, just the file name and line number. `break Directory/Path/filename.cpp:62`
break **address*	Suspend processing at an instruction address. Used when you do not have source.
break *line-number* if *condition*	Where condition is an expression. i.e. `x > 5` Suspend when boolean expression is true.
break *line* thread *thread-number*	Break in thread at specified line number. Use `info threads` to display thread numbers.
tbreak	Temporary break. Break once only. Break is then removed. See "break" above for options.
watch *condition*	Suspend processing when condition is met. i.e. `x > 5`
clear clear *function* clear *line-number*	Delete breakpoints as identified by command option. Delete all breakpoints in *function* Delete breakpoints at a given line
delete d	Delete all breakpoints, watchpoints, or catchpoints.
delete *breakpoint-number* delete *range*	Delete the breakpoints, watchpoints, or catchpoints of the breakpoint ranges specified as arguments.
disable *breakpoint-number-or-range* enable *breakpoint-number-or-range*	Does not delete breakpoints. Just enables/disables them. Example: Show breakpoints: `info break` Disable: `disable 2-9`
enable *breakpoint-number* once	Enables once