Chapter 2

Embedded programming

Contents

2.1 Multithreading and scheduling . . . **2-1**
- 2.1.1 First-In, First-Out (FIFO) scheduling . . . 2-2
- 2.1.2 Round Robin (RR) scheduling . . . 2-3
- 2.1.3 Shortest Remaining Time First (SRTF) scheduling . . . 2-3
- 2.1.4 Priority scheduling, dynamic priority adjustment, and multilevel schedulers . . . 2-3
- 2.1.5 Multicore: load balancing, processor affinity, & power management . . . 2-5
- 2.1.6 Characterizing “real time” application requirements . . . 2-6

2.2 Operating Systems (OSs) . . . **2-6**
- 2.2.1 Bare-metal programming . . . 2-6
- 2.2.2 Real-Time Operating Systems (RTOSs) . . . 2-6
- 2.2.3 Linux . . . 2-7
- 2.2.4 Realizing hard real time with Linux: dual-kernel approaches vs. RTL . . . 2-7
- 2.2.5 Android . . . 2-7
- 2.2.6 Robot Operating System (ROS) . . . 2-7

2.3 Programming languages . . . **2-7**
- 2.3.1 C, C++ . . . 2-7
- 2.3.2 Python . . . 2-7
- 2.3.3 Graphical programming environments . . . 2-7

2.4 Text editing & command-line programming versus IDEs . . . **2-9**
- 2.4.1 Command-line programming . . . 2-9
- 2.4.2 Programming in an Integrated Development Environment (IDE) . . . 2-9

2.5 Debuggable, maintainable, and portably fast coding styles . . . **2-10**
- 2.5.1 Platform-optimized libraries: BLAS, LAPack, FFTW . . . 2-10
- 2.5.2 POSIX compliance . . . 2-10

2.6 Git, github, and doxygen . . . **2-10**

2.7 Software approximation of transcendental functions . . . **2-10**

2.1 Multithreading and scheduling

A central element in the efficient use of limited computational resources for the coordination of complex electromechanical systems is *multithreading*; that is, the simultaneous running of many threads (a.k.a. *processes* or *tasks*), each at different rates and priorities, on a microcontroller with only a handful of CPU cores.

The coordination of multiple threads on a microcontroller is handled by the part of the OS called the *scheduler*. At any moment, a thread can be in one of three states: *executing* (a.k.a. *running*), *ready* (to run again, or to run some more...), or *waiting* (to be shifted back to the ready state). The component of the scheduler that shifts threads from the *ready list* to actually executing on the CPU is called the *dispatcher*.

Time-critical threads in an embedded setting generally require short periods of computation, called *CPU bursts*, followed by idle wait periods [during which file or bus i/o might be performed]. A request to the scheduler for a thread to begin a new CPU burst is initiated by some sort of *trigger* (a.k.a. *interrupt*) signal, such as

(a) a *timer*, which triggers requests for new CPU bursts on a thread at precisely predefined intervals Δt,
(b) a *delay*, which triggers such requests a set time after completion of previous CPU bursts on the same thread,
(c) a notification of the completion of a file or bus i/o (read or write) previously requested by the thread,
(d) a notification generated by the physical system, such as when a target temperature is reached, or
(e) a notification of new user input.

Other threads (e.g., video encoding) in an embedded setting are *CPU-bound* (requiring much longer computation time on the CPU), and may be worked on from time to time in the background, when the CPU bursts associated with all of the currently-triggered time-critical (higher priority) threads are complete. Note that:

(i) a running thread may shift back to the *waiting list* because its current CPU burst is complete, and the thread needs to wait for its next trigger (see above – in particular, a request for file or bus i/o is usually *blocking*, meaning that the corresponding thread is shifted back to the waiting list until the i/o is complete),
(ii) a running thread may *terminate*, simply because it finishes its task completely, or
(iii) a running thread may be *preempted* once the length of time (a.k.a. *quantum*) alotted to it is expired, or a higher-priority (time-critical) thread is triggered and needs to run, with the scheduler moving the preempted thread back to the ready list before the current set of computations in that thread complete, thus giving other threads a chance to run.

The *scheduling algorithm* (a.k.a. *scheduling policy*) is the set of rules used to determine the sequence that the threads in the ready list will be run, and the quantum that each thread is allowed to run before it is preempted to give CPU time to other threads. A scheduling algorithm must balance several competing objectives based on a limited amount of information regarding what might happen next, including:

(1) *respecting assigned priorities*: the user should be able to assign which threads are most important to complete in a timely fashion, and this preference should be enforced (thus giving "real-time" behavior – see §2.1.6),
(2) *responsiveness*: interactive threads should react quickly,
(3) *efficiency*: the CPU should be kept doing productive work all the time, minimizing the overhead involved in switching threads (see point iii above), and making maximum use of microcontroller I/O subunits (which are generally slow compared to the CPU), so that waiting on these I/O subunits does not hold up other threads,
(4) *fairness*: each thread of the same priority should receive about equal access to CPU time,
(5) *throughput*: the number of threads that accomplish something significant per second should be maximized,
(6) *avoiding starvation*: even low-priority threads should get a chance to run from time to time, and
(7) *graceful degradation*: as the CPU demands approach 100% or more, performance on all threads (particularly the lower-priority threads) should degrade gradually, and none of the threads should freeze.

Different compromises between these competing objectives are reached by different scheduling policies and different choices of the time quantums used, as illustrated by the following examples.

2.1.1 First-In, First-Out (FIFO) scheduling

To understand what a scheduler does, it is enlightening to consider first the simplest, non-preemptive First-In, First-Out (FIFO) scheduler. This scheduler simply waits for the CPU burst in the currently running thread to complete, and for the thread to enter the waiting list on its own (because, to continue, it needs to wait for a new trigger – like a timer interrupt, a notification of the completion of a blocking i/o request, etc). The dispatcher then shifts the oldest thread on the ready list over to begin executing on the CPU. Once any waiting thread receives the trigger it is waiting for, that thread is moved from the waiting list back the end of the ready list.

Though extremely simple, and effective at minimizing the overhead involved in switching threads, the FIFO approach reaches a relatively poor compromise between the seven competing objectives described in the previous section: it does not respect assigned priorities for time-critical tasks, interactive threads can be unresponsive when long CPU-bound tasks come up to run, etc. FIFO scheduling on its own is thus generally not recommended in practice (except in limited, controlled circumstances).

2.1.2 Round Robin (RR) scheduling

Round Robin (RR) scheduling amounts simply to a preemptive variant of FIFO scheduling, which improves upon the properties of the FIFO approach by limiting the quantum of time that any thread can tie up the CPU before the next thread gets a chance to run.

Using a large quantum, RR scheduling is effectively the same as FIFO scheduling, whereas using a smaller quantum results in more frequent switching between ready threads, which makes the overall system more responsive. However, reducing the quantum also increases percentage of time involved in switching threads (which typically takes a few ms), which reduces efficiency. For example, assuming a 3 ms switch time, a policy with 10 ms quantums spends $3/(10+3)=23\%$ of the time switching, whereas a policy with 50 ms quantums spends $3/(50+3)=5.7\%$ of the time switching. A compromise must thus be reached with an intermediate quantum (typically 10 to 50 ms) that provides both sufficient responsiveness and also reasonable efficiency.

2.1.3 Shortest Remaining Time First (SRTF) scheduling

Shortest Remaining Time First (SRTF) scheduling is a variant of RR scheduling that, based on historical averaging, estimates the upcoming CPU burst time associated with each thread on the ready list and, whenever the CPU becomes available, shifts the thread on the ready list with the shortest estimated CPU burst time over to begin executing on the CPU. A quantum is again used, so any thread with an actual CPU burst longer than the quantum is again preempted, and moved back to the ready list when its time is up.

A challenge with this approach is estimating future CPU burst times for any thread in the ready list, based only on previous CPU bursts in the same thread. One way to obtain such an estimate, E_n, of the n'th CPU burst time, B_n, is via an *exponential average* (a sort of IIR filter) given by $E_{n+1}=aB_n+(1-a)E_n$ for $n\geq 2$ with $0.1\lesssim a\leq 1$, where we initialize $E_1=0$ and $E_2=B_1$, with $n=1$ corresponding to the first CPU burst.

An advantage of SRTF scheduling is that it tends to move interactive tasks (with, typically, short CPU bursts) to the head of the ready list (thus improving responsiveness) and, by running the shortest tasks first, it reduces the *mean response time* of the system (that is, the average time a thread spends between entering the ready list to the completion of its corresponding CPU burst), thus maximizing throughput.

2.1.4 Priority scheduling, dynamic priority adjustment, and multilevel schedulers

To allow the user to indicate a preference regarding which threads are most important to complete in a timely fashion (objective number 1 discussed above for schedulers for embedded systems), some sort of *priority scheduling* is required. In the simplest, *static* form of such a policy, priorities are assigned (either *externally*, by the user, or *internally*, by the OS) for the life of each thread, and the threads on the ready list with the highest priority run first, with higher-priority threads preempting lower priority threads that may already be running as soon as they are moved to the ready list. In the event that multiple threads in the ready list are assigned the same priority, one of the simple policies described above (FIFO, RR, or SRTF) is used to break the tie; preempting may still be used, of course, to prevent individual threads from consuming the CPU for too long.

A clear advantage of priority-based approaches is that their behavior is easily predicted, and high-priority threads (e.g., those responsible for time-critical machine control loops, and interactive response) may be set to always run in a timely fashion, as needed. A challenge with such approaches is that some lower-priority threads may ultimately be completely starved of CPU time when the total requested CPU load exceeds 100%. To address this challenge, *dynamic* forms of this policy are sometimes used. Dynamic approaches occasionally boost the priority of some low-priority threads that haven't run in a while, thus making sure that they at least get a limited opportunity to run (this is sometimes referred to as *process aging*). Once such a boosted lower-priority thread runs for a full quantum, its priority is reduced back towards its original value. Often, longer quantums are implemented by the scheduler at lower priority levels, so with such dynamic approaches a thread can effectively settle into a priority level with a quantum that matches its typical CPU burst time, which is efficient. Note that such dynamic priority adjustments may be implemented in such a way as to never exceed the priorities assigned to the highest-priority ("real-time") threads.

It is common for a priority-based scheduler to group threads (distributed over about a hundred different priority levels) into a handful of *priority classes* [a.k.a. *priority queues*, for "real-time" (e.g., machine control) processes, system (a.k.a. kernel) processes, interactive processes, background processes, etc], each with a (possibly) different scheduling policy (like RR), and each with its own range of quantums implemented. Each of these priority classes themselves span well over a dozen priority levels, so priority-based scheduling algorithms (with or without dynamic priority adjustment) may still be used within each class. A *multilevel scheduler* may then choose to devote the CPU, when fully loaded, a certain maximum percentage of time to each class of processes, and to use a simpler priority-based scheduling policy within each class. The *Completely Fair Share* (CFS) scheduler implemented in modern Linux kernels is a general purpose multilevel scheduler implementing dynamic priority adjustment within a handful of priority classes, including a RR scheduler at the highest priority levels for "real-time" tasks.

2.1.5 Multicore: load balancing, processor affinity, & power management

SMP and HMP

Most modern embedded processors can actually run multiple threads at the same time, including:

• systems with multithreaded cores, which present themselves as two virtual cores to the scheduler, allowing multiple instructions [e.g., integer operations (IOPs) and floating-point operations (FLOPS)] to execute simultaneously on a single core, as long as they don't compete for the same resources;
• systems with multiple cores on one CPU, or with multiple CPUs, with or without shared memory caches but all with Uniform Memory Access (UMA) to all of the main memory, either:
- in a *symmetric multiprocessing* (SMP) arrangement, in which all cores are equivalent, or
- in a *heterogeneous multiprocessing* (HMP) arrangement, as in ARM's big.LITTLE and DynamIQ implementations, which combine high-performance cores (for computationally-intensive, time-critical tasks), and high-efficiency cores (for simpler, lower-priority tasks);
• systems with multiple CPUs in a Nonuniform Memory Access (NUMA) arrangement, in which each compute core has a certain portion of the main memory closely affiliated with it, and thus can reach some parts of the it faster than others (such systems may also be SMP or HMP) - embedded processors with large GPU-based computational subsystems, and those with dedicated "Neural Processing Units", are typical examples.

The same general considerations and scheduling policies discussed previously still apply in these settings, but now with the complex additional consideration of needing to manage the delicate question of which core should be used to run a particular thread next, and which cores can be run at reduced clock speeds, or powered down entirely, during relatively idle periods of time in order to save power.

These delicate questions need to be handled carefully by modern schedulers in order to balance computational throughput and power efficiency in the system, and very different solutions are needed for servers, laptops, cellphones, and microcontrollers for "real-time" control of embedded systems. Notably, balancing computational performance with power efficiency is becoming increasingly important in all types of computational platforms, and solutions originally developed for small battery-powered systems (cellphones) are working their way up to laptops and large server farms, which are increasingly limited by power considerations.

In multicore settings, the issues of *processor affinity* and *load balancing* must be addressed. That is, it is usually much more efficient to run a new CPU burst on the same core (or at least on the same CPU) that a thread ran on previously, because the memory cache corresponding to that core (or CPU) is probably already set up with much of the data that that thread needs to run again. However, sometimes threads still need to be shifted from one core to another in order to balance the load across multiple cores in the system, as the scheduler seeks to maintain its target balance between computational throughput and power efficiency. Hierarchical *scheduling domains* are often introduced in order to handle these questions, with lower-level schedulers handling each individual core, and higher-level schedulers occasionally moving threads from one core to another as necessary (i.e., whenever a given core becomes relatively overloaded, or underloaded, with tasks to complete). To improve the predictable performance of the most time-critical threads, including those that might share certain cached data, it is often beneficial to implement *hard processor affinity* for such threads, binding them permanently to specific "reserved" cores, while allowing the OS to manage the other threads that might come and go on the system (but possibly restricting the other major threads on the system from running on the reserved cores, thereby preventing them from interfering with the most time-critical processes).

Thankfully, the complex coupled problems of scheduling, load balancing, and power management for SMP and HMP multicore systems are generally taken care of by the OS, not by the embedded programmer, and the sophistication with which modern schedulers for multicore systems address these problems, to appropriately balance computational performance with power efficiency, is evolving rapidly. However, understanding generally how such schedulers work is essential for the embedded programmer, in order to select and use a scheduler appropriately, and to tweak its behavior effectively (in particular, to set priorities correctly, and to use hard processor affinity where appropriate), in order to strike the desired balance between the seven objectives outlined previously: namely, to get sufficiently reliable "real-time" performance for high-priority time-critical tasks (see §2.1.6), sufficient responsiveness from interactive tasks, and efficient performance on all other threads that the computational system needs to manage, even as the computational system becomes fully loaded.

2.1.6 Characterizing "real time" application requirements

In §2.1, the #1 objective listed for a scheduler on a microcontroller is that the user should be able to assign which threads are most important to complete in a timely fashion, and that these preferences should somehow be enforced. In embedded systems, we need to define the importance of such preferences with precision. Consideration must first be given to the application itself. Embedded programmers often catagorize controllers based on the consequences of not completing a task within a specified time constraint (a.k.a. deadline):

• *hard real-time* controllers are designed for systems in which a missed deadline may result in total system failure [an assembly line shuts down and needs to be physically repaired, a rocket blows up, ...];
• *firm real-time* controllers are designed for systems in which, after a missed deadline, the utility of a result is zero [a single part will be rejected (automatically) off an assembly line, a toy falls over, ...]; and
• *soft real-time* controllers are designed for systems in which, after a missed deadline, the utility of a result is reduced somewhat [an RC car is momentarily unresponsive to user input, a hamburger bun is slightly singed, ...].

In addition to specifying the relevant deadlines themselves, the above characterizations of the consequences of missed deadlines are valuable when deciding how to allocate limited computational resources to potentially complex electromechanical systems.

Hard real time requirements are actually somewhat rare in well-designed mechanical systems; examples might include the control of an unstable chain reaction, or a pacemaker for a human heart. In hard real-time systems, particularly those that are safety-critical, mathematical guarantees of no missed deadlines are often required. Guaranteeing such hard real-time behavior is generally only possible by applying a controller in a relatively isolated setting with simple (and, thus, highly predictable) bare-metal programming (see §2.2.1), without several other threads running simultaneously that might occasionally through the timing off.

More often than not, however, threads running on embedded systems call for firm real-time and/or soft real-time behavior. In such systems, the priority-based preemptive scheduling strategies described above, as implemented by a well-designed OS (e.g., the PREEMPT_RT patch of the Linux kernel) and used properly by a careful programmer, are most often entirely sufficient.

2.2 Operating Systems (OSs)

2.2.1 Bare-metal programming

In this section and the two that follow, we outline the three fundamental programming paradigms for embedded systems, in order of simplicity.

Arduino
ladder logic
programmable logic controllers (PLCs) used in industrial control applications

2.2.2 Real-Time Operating Systems (RTOSs)

nuttx (posix compliant)

keil RTX
Real Time Executive for Multiprocessor Systems (RTEMS)
real time linux
more realtime linux
Case study: FreeRTOS

2.2.3 Linux

2.2.3.1 Embedded Linux distros

distros (distributions)
Debian (derivatives: Ubunto, Raspberry Pi OS).
Yocto. OpenWrt.
Commercial: Wind River Linux. Red Hat Embedded
Look for lightweight IoT version (but, man pages are useful...)
shells

2.2.3.2 Chmod

2.2.3.3 Makefiles

real time computing

2.2.4 Realizing hard real time with Linux: dual-kernel approaches vs. RTL

PREMPT-RT
NTP service

2.2.5 Android

2.2.6 Robot Operating System (ROS)

2.3 Programming languages

Many programming languages are growing in importance in different aspects of robotics, including CUDA for GPU programming, TinyML for machine learning,

2.3.1 C, C++

2.3.2 Python

2.3.3 Graphical programming environments

Scratch
Simulink
Labview

ping 192.168.8.1	measure speed of connection to machine with IP number 192.168.8.1
ssh 192.168.8.1	securely open a shell on 192.168.8.1
echo $0	show what kind of shell you are currently in
uname -a	show info about processor architecture, system hostname, and kernel version
zsh or bash ; exit	spawn & enter a new zsh or bash shell (inside current shell); exit this shell
chsh -s /bin/zsh	change your default shell to zsh (recommended, if it isn't already)
pwd	print the name of the current working directory
ls -lah	list all files in current directory, including ownership, privileges, and size
mkdir foo	make a new directory named foo
cd foo ; cd ..	change directory to foo; change back to parent directory
touch bar	create a new file named bar (or, just update its timestamp)
echo 'hello' > bar	create (or, erase and create) the file bar, and write "hello" to this file
echo 'world' >> bar	append "world" to the file bar (or, create and write to this file)
man echo ; (space) ; q	display detailed manual page (alternative to Google) for the command echo
cat bar	show contents of the file bar (all at once)
less bar ; (space) ; q	show contents of the file bar (pausing after each screenfull)
head bar ; tail bar	show the 10 lines at the head (or, the tail) of the file bar
(up arrow) ; (down arrow)	scroll up to recently executed commands; scroll down
history	show a list of recently executed commands
hist (tab)	complete (as far as possible) name of command(s) starting with "hist"
rm bar	remove (warning: permanently!) the file named bar
rmdir foo	remove directory foo, but only if it is empty
rm -rf foo	remove recursively the directory foo and all files contained in it (danger!!!)
cp foo/bar* foo1/.	copy all files starting with the letters bar in foo into the directory foo1
cp -r foo foo1	copy recursively everything in foo to the directory foo1
mv bar foo/bar1	move and rename the file bar as bar1 inside the directory foo
chmod 644 bar	change mode (§2.2.3.2) of bar to read/write for owner, read for group & world
chown foo1:foo bar	change ownership of file bar to user foo1 and group foo
sudo rm bar	do the command rm bar as superuser (danger!)
su ; exit	enter superuser mode for subsequent commands (danger!!!); exit su mode
df -h	report disk free space on the available filesystems
du -sh foo	report significant disk use within directory foo
grep psfrag *.tex	search files ending in .tex (in current directory) for the string "psfrag"
top	periodically report a list of all running threads, sorted by top CPU usage
ps -ef	report all running processes (once)
ps -ef |grep kernel	pipe output of ps to grep, to extract the lines with "kernel" in them
file bar	test bar to determine what type of file it is
find bar	scan current directory and all its children for filenames containing bar
tar cvfz fb.tgz fb	compress all contents of fb into a (compact) gzipped tarball fb.tgz
scp fb.tgz bar:.	securely copy fb.tgz to machine with name bar on local network
tar xvf fb.tgz	extract contents of fb.tgz, retaining its original directory structure
alias l='ls -lah'	use "l" as a shorthand alias for the command "ls -lah" in this shell
env	list all aliases and other environmental variables defined in this shell
vim ; nano	command-line text editors (see §2.4.1.2) available in all linux distros
~/.bashrc	initial run commands executed when a bash or zsh shell is spawned
make foo	run commands in Makefile (see §2.2.3.3) to make an executable boo

Table 2.1: Some essential linux and unix commands (in zsh and bash). Explore! You'll find your way quickly...

2.4 Text editing & command-line programming versus IDEs

2.4.1 Command-line programming

2.4.1.1 Workflow: edit locally, sync files with SBC, compile, link, run, rinse, repeat

2.4.1.2 Command-line editors (vim and nano) vs text editors

vim and nano

text editors:

Sublime Text

Atom

Notepad++

2.4.1.3 Command-line scp/sftp/rcp vs FTP Clients

SFTP

FileZilla

2.4.2 Programming in an Integrated Development Environment (IDE)

Eclipse,

STM32CubeIDE (for STM32 devices, based closely on Eclipse)

ARM Keil MDK (for ARM devices)

Visual Studio Code (Microsoft),

NetBeans

Code::Blocks

CodeLite

Qt Creator,

PyCharm (for Python),

MPLAB X (PIC, AVR)

2.4.2.1 Workflow: debug directly within the IDE

Case study: Eclipse

Version of Eclipse for STM32CubeIDE

2.5 Debuggable, maintainable, and portably fast coding styles

self-optimizing compilers

2.5.1 Platform-optimized libraries: BLAS, LAPack, FFTW

2.5.2 POSIX compliance

2.6 Git, github, and doxygen

doxygen

2.7 Software approximation of transcendental functions

Significant attention has been put into developing efficient and accurate numerical approximation of transcendental functions (sin, cos, tan, atan, exp, ...). The definitive text on this subject, which presents all of the commonly needed (complicated-to-derive, yet simply-to-use) formula, based on truncated Chebyshev and Bessel series expansions with tabulated coefficients, is Hart (1978), a few results of which are summarized below. As mentioned previously, in embedded applications, we are primarily interested in half precision and single precision applications, which form our focus here.

The following formula (which may be computed using single precision arithmetic) approximates $\cos(z)$ over the range $0 \le z \le \pi/2$ to about 3.2 decimal digits (appropriate for use in half precision applications):

$$c_1 = 0.99940307, \quad c_2 = -0.49558072, \quad c_3 = 0.03679168 \quad \Rightarrow \quad \cos(z) \approx c_1 + z^2(c_2 + c_3\, z^2), \tag{2.1}$$

and the following formula (which may be computed using double precision arithmetic) approximates $\cos(z)$ over the range $0 \le z \le \pi/2$ to about 7.3 decimal digits (appropriate for use in single precision applications):

$$\begin{aligned} &c_1 = 0.999999953464, \quad c_2 = -0.4999999053455, \quad c_3 = 0.0416635846769, \quad c_4 = -0.0013853704264, \\ &c_5 = 0.00002315393167 \quad \Rightarrow \quad \cos(z) \approx c_1 + z^2(c_2 + z^2(c_3 + z^2(c_4 + c_5\, z^2))). \end{aligned} \tag{2.2}$$

Note the tradeoff: the first approximation is simpler (smaller table of numbers and faster to compute, but less accurate), while the second is more complex (larger table of numbers and slower to compute, but more accurate). This tradeoff is evident in all such approximations. To extend the range to $-\infty \le x \le \infty$, note that

$$\cos(x) = \begin{cases} \cos(y) & \text{if } q = 0, \\ -\cos(\pi - y) & \text{if } q = 1, \\ -\cos(y - \pi) & \text{if } q = 2, \\ \cos(2\pi - y) & \text{if } q = 3, \end{cases} \qquad \text{where} \qquad \begin{aligned} &c = \text{floor}(x/(2\pi)), \\ &y = x - 2\pi c, \quad (\text{and thus } \; 0 \le y \le 2\pi), \\ &q = \text{floor}(y/(\pi/2)) \in \{0, 1, 2, 3\}, \end{aligned} \tag{2.3}$$

where floor denotes integer division (rounding down), and $\cos(z)$ may be approximated (in any of the four cases) using (2.1) or (2.2). Applying (2.3) in order to extend the approximation (2.1) or (2.2) to the larger range $-\infty \le x \le \infty$ is called **range reduction**. With the above formulas, $\cos(x)$ and $\sin(x) = \cos(x - \pi/2)$ may be computed efficiently for half and single precision applications for any real x.

Similarly, the following formula (which may be computed using single precision arithmetic) approximates $\tan(z)$ over the range $0 \le z \le \pi/4$ to about 3.2 decimal digits (appropriate for half precision applications):

$$c_1 = -3.6112171, \quad c_2 = -4.6133253 \quad \Rightarrow \quad z_0 = 4z/\pi, \quad \tan(z) \approx c_1\, z_0/(c_2 + z_0^2), \tag{2.4}$$

and the following formula (which may be computed using double precision arithmetic) approximates $\tan(z)$ over the range $0 \le z \le \pi/4$ to about 8.2 decimal digits (appropriate for single precision applications):

$$\begin{aligned} &c_1 = 211.849369664121, \quad c_2 = -12.5288887278448, \quad c_3 = 269.7350131214121, \\ &c_4 = -71.4145309347748 \quad \Rightarrow \quad z_0 = 4z/\pi, \quad \tan(z) \approx z_0\,(c_1 + c_2\,z_0^2)/(c_3 + z_0^2(c_4 + z_0^2)). \end{aligned} \tag{2.5}$$

To extend the range to $-\infty \le x \le \infty$, note that

$$\tan(x) = \begin{cases} \tan(y) & \text{if } o = 0, \\ 1/\tan(\pi/2 - y) & \text{if } o = 1, \\ -1/\tan(y - \pi/2) & \text{if } o = 2, \\ -\tan(\pi - y) & \text{if } o = 3, \\ \tan(y - \pi) & \text{if } o = 4, \\ 1/\tan(3\pi/2 - y) & \text{if } o = 5, \\ -1/\tan(y - 3\pi/2) & \text{if } o = 6, \\ -\tan(2\pi - y) & \text{if } o = 7, \end{cases} \quad \text{where} \quad \begin{aligned} &c = \text{floor}(x/(2\pi)), \\ &y = x - 2\pi c, \quad (\text{and thus } \; 0 \le y \le 2\pi), \\ &o = \text{floor}(y/(\pi/4)) \in \{0,1,2,3,4,5,6,7\}. \end{aligned} \tag{2.6}$$

The following formula (which may be computed using double precision arithmetic) approximates $\text{atan}\,(z)$ over the range $0 \le z < \text{atan}\,(\pi/12)$ to 6.6 decimal digits (appropriate for single precision applications):

$$c_1 = 1.6867629106, \quad c_2 = 0.4378497304, \quad c_3 = 1.6867633134 \quad \Rightarrow \quad \text{atan}\,(z) \approx x\,(c_1 + x^2\,c_2)/(c_3 + x^2). \tag{2.7}$$

To extend the range to $-\infty \le x \le \infty$, define $c = \tan(\pi/6)$, and apply any or all of the following identities

$$\begin{aligned} \text{atan}\,(x) &= -\text{atan}\,(-x) && \text{if } x < 0, \\ \text{atan}\,(x) &= \pi/2 - \text{atan}\,(1/x) && \text{if } 1 < x, \\ \text{atan}\,(x) &= \pi/6 + \text{atan}\,[(x-c)/(1+cx)] && \text{if } \tan(\pi/12) < x \le 1. \end{aligned} \tag{2.8}$$

Defining $c = 2.75573192e-6$, the following formula (derived from a simple Taylor series) approximates $\exp(z)$ over the range $-1 \le z < 1$):

$$\exp(z) \approx c\,(362880{+}x\,(362880{+}x\,(181440{+}x\,(60480{+}x\,(15120{+}x\,(3024{+}x\,(504{+}x\,(72{+}x\,(9{+}x))))))))). \tag{2.9}$$

To extend the range, one may define $c = \tan(\pi/6)$, and apply ....

Simple Matlab codes that demonstrate the above several formulas (cos_32.m, cos_73.m, tan_32.m, tan_82.m, atan_66.m) are available at github site for this text. Any practical (fast) embedded application must rewrite these codes in C.