INTRODUCTION

METAPHORS
how to visualize
the computer
events
story telling
pattern matching
feedback
blocking
mapping
graphs
code execution
data/memory

TECHNIQUE how to do it -- technique constructs the strategy
events
loops and loop()
subroutines
sequence
state machine
blocking
program layout

TBD “classical algorithms” (knuth, newton's method of recip v2p264)
representation of numbers
modular arithmetic
multiple precision
precision
randomness
data structures
cacheing

program layout
work flow
style

CASE STUDY: FANCY BLINKER
CASE STUDY: CAR COMPUTER SYSTEM

TROUBLE-SHOOTING STRATEGIES

SIMPLE CIRCUITS

copyright Tom Jennings  tom@SensitiveResearch.com
INTRO

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop." -- Lewis Carroll, Alice in Wonderland

this is a very rough work in progress. it will expand into a complete work of some sort in 2016. the basic cores of metaphor, strategy and technique here are place holders, which i'll inflict on students in CalArt's Art & Technology program's Algorithmic Arts classes 1 (fall 2015) and 2 (spring 2016). experience from using this in that context will be applied to this document.

feedback and corrections appreciated.

it is more or less required that you read the first three chapters, as an absolute minimum, of PHILOSOPHY IN THE FLESH by Lakoff and Johnson for a grounding in the concepts of metaphor used here. the first three chapters are an easy read, but i suggest you get most of the way through it.

prior knowledge required (crib knuth v1 preface)

construct frameworks with which to see and create.

summarize metaphor, strategy, technique.

strategy broadly lays out the overall approach, the ideology of events, time, sampling, and blocking.

technique instantiates the strategy

algorithm: a definite method

events are information

code is fast, the world is slow.
METAPHORS
METAPHORS

you will have a difficult time writing code without some way to visualize what goes on inside a computer.

it is no coincidence that the development of computer programming languages (once called “high level languages”) eventually stumbled upon the same concepts described as basic categories within the science of embodied cognition\(^1\). the history of computer programming language development has long been dominated by issues of scope and complexity. programming language semantics of code and data block structure exactly parallel Lakoff and Johnson's containers. control-flow statements have many of the attributes of paths. metaphors of memory as space are widely understood. these ideas, and more, will be discussed in the pages below.

there is no escape from using metaphors to describe computers because there are few things of any complexity in the world that we can describe without using metaphors grounded in human bodily experience in some way. in this document i use metaphors explicitly to describe computer components and systems, externally attached electronic devices, the CPU executing programs. metaphors dominate strategic approaches to problem-solving with a computer, as they must since computer operation is utterly invisible. many of the metaphors here i absorbed through decades of programming experience, but i have attempted to pass them all through the filter of Lakoff and Johnson’s succinct writing on embodied cognition.

THE “COMPUTER” MODEL

there has long been a de facto universal model used to represent computers, first uttered in it's modern form by Charles Babbage in the 1830's and formalized by Alan Turing in 1936.

Babbage described his computer as having two distinct but inter-dependent units, the mill and the store. the mill takes numbers from the store, performs calculations, and places the result back in the store. the metaphor for these were once contemporary: the grist mill a complex machine that takes in raw harvested wheat, grinds it, and puts the resulting flour back into storage.

Turing's computer was a thought machine consisting of an infinite linear tape marked into identical squares. each square may contain a written symbol. the machine is free to move back and forth along the tape and make decisions (erase, write, move) based upon symbols read from the tape.

this machine/memory duality is common to both. the machine/mill read instructions that the machine interprets and acts upon, and those instructions operate on the symbols contained in the mill/tape. each model approximates the (very Cartesian) relationship of an intelligent entity (mill/scanner) interacting with a world of symbols and actions (store/tape). turing explicitly models his machine as a

\(^1\) i strongly recommend “Philosophy In The Flesh: the Embodied Mind and its Challenge to Western Thought” (Lakoff, George and Johnson, Mark,1999); the first three chapters, 45 pages, will get the point across.
“we may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions... the machine is supplied with a 'tape' (the analogue of paper) running through it, and divided into sections (called 'squares') each capable of bearing a 'symbol'...”

mill/machine and store/tape are interdependent; neither has purpose without the other. each are opposite and complementary. neither more important than the other, each is useless without the other.

we have a strong tendency to view anything of sufficient complexity or behavior, constructed or “natural”, as a sentient entity. sailing ships, human/machine systems of extraordinary complexity, are even gendered. the modest CPU in the Arduino microcontroller contains millions of components most maintaining independent state, lending it enough innate complexity to “feel like” an entity.

please do not read into this that the mill/machine is in an active agent with power over passive data. the machine doesn't exist in a vacuum and the difficulty of writing effective software coupled with an endless supply of unexpected side effects more accurately indicates precisely the opposite -- that the entity must work hard to make sense of any thing at all. just because the machine/entity cannot “see” agency behind detected stimuli doesn't mean that it is not there -- each datum may be operated on by seen and unseen entities!

the ideology of duality persists in all metaphors of computing and all of this writing should be seen in this light.

EVENTS

About-to-act is an interesting state to experience, because i am conscious of just those tensions. Acting itself feels fairly dull ... Acting is only interesting as it leads to new tensions that, irrelevantly, cause me to act again.

kidd, in Samuel Delany's Dhalgren, 1975.

how to visualize

the concept of the event is the single most important idea in effective real-time computing.

an event is something that happens and that is of concern to us. events are conceptual things that manifest in a computer as a change in some datum -- a digital input pin, an analog input pin, the result returned from a subroutine or function, or the contents of a memory variable.

• a change is an event if we consider it significant
• if we consider a change as significant we treat as an event

events are information. information is a change in state of matter\(^3\). there is no information that does not involve mass (however miniscule).

events are changes in state. in an event-driven strategy, events are interesting; stasis is uninteresting. change commands our attention, and large chunks of our cognition are devoted to detecting change. it's what we do. things that do not change do not require any attention.

in a house a door is just a door. a door, open or closed, is not generally "interesting". when a door opens suddenly it changes state; at that moment the door may become interesting -- if at that moment we are concerned with doors. we have visual cortex devoted solely to noticing such changes in the world.

the concept of change is more complex than it seems. it requires remembering the past; if you can't recall the door's previous state you cannot know if it has changed. we cannot articulate anything about events without binding them to time. a change in state requires memory of a previous state. we take memory for granted, but in a machine, remembrance must be made explicit.

you could argue that the static state of a thing (a door, a light switch) is in fact important. that may be true but is missing the point. an event in a computer is as much ideology as physical fact. events signal that something requires action -- if it did not, we would not consider the change to be an event. much

\(^3\) C. E. Shannon, *A Mathematical Theory of Communication*, Bell Technical Journal July 1948, p18. “If a source can produce only one particular message its entropy is zero, and no channel is required. For example, a computing machine set up to calculate the successive digits of \(pi\) produces a definite sequence with no chance element. No channel is required to “transmit” this to another point. One could construct a second machine to compute the same sequence at the point ....”
change in the world is ignored because we *decide* to ignore it. the fact that a thing is unchanging (or ignored) means that it no longer *requires our attention*. in this framework any “static” state of existence can be re-created by paying attention to only changes in state -- the event of a wall switch state (position) change means that action must be taken to change the state of a lamp, accordingly.

change in state creates information and consumes energy.

strategic frameworks are ideologies. they intentionally introduce constraints on what and how we perceive in order to narrow (simplify) the decisions we make to effect some desired outcome. the strategic framework you choose should enhance your ability to work in the world.
TIME AS AN EVENT

how to visualize

Time is an illusion
perpetrated by the manufacturers of space.
-- GRAFFITI

time in this context means elapsed time, not 12 or 24 hour wall clock nor calendar time. we can
shortcut deep philosophical questions about the nature of time, because in the Arduino (and most
computers) time is a simple number line of ever-increasing positive integers.

elapsed time is sometimes needed while cooking. a common technique to measure a passage of time
is to read the wall clock, add the desired number of minutes to that reading, then await that future
time. code to blink an LED uses precisely this same technique, reading the internal “clock” (which
counts milliseconds), adding the desired time interval to that reading, saving it, and periodically
checking for the clock to reach or surpass the new saved time. timed events are predictable numbers
that "come up" on the number-line of time in the computer.
STORY TELLING

*why we do it this way*

"The only important elements in any society are the artistic and the criminal, because they alone, by questioning the society's values, can force it to change."

-- the Linguistic Ubiquitous Multiplex (Lump)
in EMPIRE STAR by Samuel R. Delany

events and resulting acts/actions, especially those that unfold over time, are in essence stories. the story is a deep and important means of communicating richness and complexity. good story telling is a substantial art, and code-writing especially within inter-active, reactive event-driven computing machinery is certainly story telling.

story-telling doesn't require narrative; it can be as simple as the unfolding of one minor act in a larger story of any kind. theater, writing, media installations, interactive or not, are all forms of story telling and being aware of this as you code will make for better results.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX PLACEHOLDER
PATTERN MATCHING

*how to visualize*

if i had to describe human cognition in one phrase it might be: pattern matching. patterns are a deep low-level concept and construct. finding and creating patterns is what metaphor does. pattern matching simplifies the world for us; once you've figured out how “door” works you more or less understand anything door-like and can see anything with a certain set of properties as “door like”. this is not unique to humans; all mammals, at least, do it. dogs understand “doorness” and easily distinguish a door from a wall with similar visual characteristics.

we see patterns everywhere, even when they don't exist⁴.

the use of patterns in computers permeates everything from the hardware, code we write, the data and events we deal with, and importantly, the way we write and structure code.

---

FEEDBACK

how to visualize

in a similar vein, if I had to absurdly reduce the scientific accomplishments of the 20th century to one word, it would be feedback\(^5\), succinctly defined as *mechanistic circular causality* on Wikipedia. A computer is the ultimate (so far) feedback machine. Though the rudimentary concept of feedback has been in use for centuries -- ancient Greek water-flow regulation, the flying-ball governor of early steam engines\(^6\), -- as a concept in electronics it was struggled with (and against) through the 1940's. The field of cybernetics, the study of humans and machines in closed-loop feedback systems\(^7\) is based upon it. Cheap and ubiquitous computers have turned a once-arcane subject into a common everyday technique.

Of course many computer applications don't use or require feedback. Many or possibly most “desktop” programs do not appear to be feedback-based; web browsing, writing, drawing, etc. come to mind. But in fact those nearly always involve feedback through the human user, eg. When drawing your eye informs your hand as you draw or type; you rarely type for very long without looking at the screen. Computer games are more obviously feedback-based. Feedback where human activity in the loop is the very definition of *cybernetic*.

Most embedded controller applications however involve feedback of some sort, either implicitly or explicitly, and that is our main concern here.

feedback involves a closed-loop of circular causality. Our computer accepts information on its inputs, (either as event or continuous measurement); our code within directs decisions based upon that input.

---


6  https://en.wikipedia.org/wiki/Centrifugal_governor


ROUGH DRAFT/PROTOTYPE copyright Tom Jennings  tom@SensitiveResearch.com
and on *a priori goals embedded in the code itself*, and outputs changes to the real-world process to affect though changes. the results of those changes are sensed as changes to the inputted information.
BLOCKING

why do it this way

blocking is a critical concept in how we talk about reacting to events and carrying out actions in response to events. blocking is easy to understand because it’s a common human experience. writing code to deal with blocking means fully understanding assumptions made when we deal with things that block our progress in the real world.

we encounter blocking daily (and we sometimes block other people when we’re not paying attention). after waiting your turn at a take-out coffee counter, you place your order with the server and move aside to wait, allowing others behind you place their order. if you remain standing at the counter you would block everyone behind you. waiting in line does not get your coffee any quicker than standing to one side, and as an unpleasant side effect wastes everyone else's time.

cooking a meal might require you to fry vegetables and boil noodles. you can do those tasks sequentially; if the former takes 9 minutes and the latter 6, cooking takes 15 minutes total. but with little extra effort, you can do both at the same time, alternating your attention between the two tasks, and accomplish both in the time it takes to complete the slower task, 9 minutes in this case.

we will examine this metaphor more closely, because it contains giant hints as to how you can arrange code to not block.

• first, you probably have a decent understanding of the two processes. though i said frying vegetables takes 9 minutes boiling noodles 6 it doesn't matter much if those estimates are wrong as long as they are not crazy-wrong (eg. 400 minutes to fry, etc). it doesn't matter if one completes before the other, as long the total time is reasonable.
• second, while it takes minutes to actually cook food, it takes almost no time to switch your attention from one pan to another. in other words, you can respond to changes much faster than the food itself changes. this is a critically important point -- trivially obvious to us while cooking but requiring care in software. code is fast, the world is slow.
• third, to state the obvious, you know that while you are looking at one pan, you are pointedly not looking at the other. you can do this because food cooks slowly relative to your ability to switch attention.
• last, you know that if something distracts you from either task (talking on the phone, etc) you will end up with burned vegetables or mushy noodles. distraction equals blocking.

our strategic approach to solving (software) problems is very analogous to the above. all of our metaphors are grounded in the body. few things escape this fact (though quantum physics is one). even mathematics and logical bit-manipulation inside complex data structures can be (must be) visualized in some way, for us to be able to think about it at all.

ROUGH DRAFT/PROTOTYPE   copyright Tom Jennings   tom@SensitiveResearch.com
DATA MAPPING

to *map data* is to take a quantity from one concept and applying it to another concept. data mapping is analogy-making in code. the name derives from the process of making maps (of the earth). a map is not the territory -- a black line on paper is not a road, a blue line is not water.

mapping data is algorithmic metaphor production. it is the innermost heart of good program design. it is best and easiest done intentionally.

it is often difficult to state precisely how it is that we map things, but good mappings feel quite natural. we do it all the time and good ones are immediately useful. frozen or boiling, water is neither red nor blue. we import metaphors from other experiences and sensations; though the ocean is blue (from reflected sky) and often cold, heater water is never red, though fire sometimes is.

all mappings depend on the meanings of symbols defined by culture. there are no universal symbols or meanings, but those that stay close to the body are more easily learned. the Lakoff and Johnson's *basic categories* describe the mappings underneath many metaphors; high is up is hot is elevated temperature; low is down is turn down the heat.

mapping and representations of physical quantity sometimes blur; we can casually speak of the brightness of an LED as low to high, and represent this as a number, 0 to 255, and never be aware of the fact that all of it, including the rapidly changing electrical states produced at the interface pin, is all metaphor and analogy. in no way is “255” inherently the same thing as “bright”, but countless humans over decades of time with the same bodily cognitive apparatus collectively, mostly unconsciously, arranged for it to be so.

in software, you can make your own mappings, but it behooves you to know why they work.

the HSV system (Hue, Saturation, Value) system of representing color within a computer is an example of mapping that at base is extremely simple. each of the components is a number that runs from 0 to some maximum (N). Value means brightness; 0 is none (“dark”) N is maximum brightness.

---

8  underneath? is there anything of importance that can be said without metaphor? ’i was hungry then i ate food that tasted good’ hardly describes a delicious dinner experience.

9  https://en.wikipedia.org/wiki/HSL_and_HSV

ROUGH DRAFT/PROTOTYPE  copyright Tom Jennings  tom@SensitiveResearch.com
hue is interesting. 0 is mapped to red, and as the number increases color passes through yellow to orange to blue to violet and approaches red again as it reaches maximum N. therefore hue is a continuum. it “looks good” though it is an entirely made-up system with little basis in physical or biological reality. the fact that it is a fiction in no way detracts from its usefulness.

your ability to write meaning into code will be limited only by your ability to come up with workable mappings. bad mappings are the major source of “what is it supposed to do?” arm-waving in computer-mediated interactive installations.

EVENTS AS MAPS

maps are usually both concrete and symbolic in some way. maps are synaesthetic simultaneity. red is hot now, 11 is loud now. maps are static, in that they aren't experienced as unfolding over time. maps are sensed, “passively”, they don't involve your body moving and expending energy in and of themselves.

events are a form of mapping also, but have a different temporal arrangement: events are causal,
triggering an action or series of actions to unfold, now and/or ongoing. Events involve your physical body, at least potentially. But events also are metaphorical, and actions that follow from an event are a form of “story telling”.

The flexibility of the technique of mapping allows us to turn (for example) color into a number and that number, algorithmically, into another color -- or a sound if we desire it. Mapping is the basis for a fundamentally important visualization tool, the graph. Written on paper or produced by machine, graphs are an utterly required visualization tool.
Observation, and not old age, brings wisdom.
-- ANONYMOUS

the graph is a critically important visualization tool and technique. a graph creates information as it is made, most commonly for revealing patterns in change of some quantity over time (though any pair of datums can be graphed on flat paper). we are experiential animals. we see change in some quantity as it occurs in real-time (human time) and from that infer and anticipate change -- which is precisely the problem; the feeling that we understand what we are seeing blinds us to the narrowness of our vision.

the graph is a critically important visualization tool and technique. a graph creates information as it is made, most commonly for revealing patterns in change of some quantity over time (though any pair of datums can be graphed on flat paper). we are experiential animals. we see change in some quantity as it occurs in real-time (human time) and from that infer and anticipate change -- which is precisely the problem; the feeling that we understand what we are seeing blinds us to the narrowness of our vision.

graphing a change in some quantity over time does a number of things at once. first, it frees us from the limitations of organic memory and attention span -- in the world of electronics and code time scale varies widely. the electrical/mechanical activities of a simple switch unfold in millionths of a second; cycles of ambient light unfold over hours or days. we have neither the speed nor glacial patience to assimilate these experiences in our minds. second, your eyes miss much of what is actually happening -- a scalar value changing moment by moment describes a shape you cannot yet see until it is graphed. third, our internal experiences are invisible to other people -- you need to graph (draw, write, etc) to communicate or collaborate with others. a two-dimensional graph is a near-universal communication tool. fourth, events are ephemeral; plots and graphs are forever. you can compare a graphed event today to a similar one yesterday, or one that may happen tomorrow. fifth, to write computer programs we need unambiguous data to insert in our code.

graphs are a visual tool for thinking about events and actions and relationships. graphs make evident interrelationships that you intend to code for. they are communication tools for your ideas and intentions.

there is artfulness in graphs. it is often an austere beauty, but a good graph can reveal much about a vision of the world.
DRAWING GRAPHS

A graph is produced over time and draws a visual shape of change, a tangible record of an experiential event. The underlying visual model of a graph is a long roll of paper moving from right to left, representing elapsed time. Imaging yourself holding a pencil to the paper as it scrolls beneath. You move the pencil only vertically, the position a metaphor for the quantity to be measured; low to high.

You only need concern yourself with the (vertical) quantity at the moment it is experienced. The horizontal motion of the paper and the vertical motion of the pencil combined draw the graph over time.

The resulting whole is more than sum of the inputs, the unfolding series of actions of the past is now bound timelessly to paper.
an oscillograph is an electronic instrument that draws some variable, X, vs. time graph onto an LCD screen; the horizontal time scale, Y, can vary from seconds to picoseconds and record events well outside our ability to sense. oscillographs are wonderfully useful devices; alas, due to the fact that until recently they were very expensive the culture around them remains concentrated in large professional shops. they are however amazing tools that reveal countless hidden worlds in anything electrical or electronic.
CODE EXECUTION (CPU) METAPHORS

I cannot conceive that anybody will require multiplications at the rate of 40,000, or even 4,000 per hour; such a revolutionary change as the octonary scale should not be imposed upon mankind in general for the sake of a few individuals.
-- F.H. WALES (1936)

a useful metaphor for the CPU in a computer is the *demon* -- in the Greek mythological sense of an independent, super-natural entity, with no particular bias towards good nor evil. the origin is well known: *Maxwell’s demon*, a thought experiment (metaphor) created by physicist James Clerk Maxwell\(^\text{10}\) in the 1860’s.

the demon/CPU reads your code/script and does exactly what the code says to do -- which may not be precisely what you meant. things that go wrong are often surprising, difficult to puzzle out, and when the code has control over external hardware -- motors, servos, lights, etc -- can result in actual physical harm (sometimes dramatic). because code is “mere” description, it can sometimes be hard to tell when things *go right*. trouble-shooting -- debugging -- is an art unto itself and while almost entirely subjective and experience/skill based it is amenable also to a strategic approach; see section XXXXXXXXXXXXXXXXXXXXX for hints.

the entire point of having a functional software *strategy* is for us fallible humans to have a clear, concise and practical method for en-coding (writing) *what we want to occur* in software.

here are some additional ways to think about code execution. the CPU *reads* your program statement by statement, *executing* each statement in order. the CPU follows your program as you might follow a recipe. the steps executed in a program follow a *path* that varies due to control statements such as *if...then*. programs/paths always have a beginning, while most desktop-type programs have an end and may terminate, Arduino computers run a single program from power-on until power-off, consistent with their embedded environment.

analogous to the kind of paths that humans follow, the path of execution has the following characteristics. at the outset you have a goal and a nominal path to that goal. no matter what steps you may take you always remain on the path, because the steps you take is the path. some steps in the path may involve choosing the next step; one iteration may be to the left, and the next to the right. just as a friend might give you driving directions that appear sensible at the outset but during execution unpredictably make no sense at all, computer programs can do truly unexpected things (but they always do what you *told* them to do).

an old metaphor used when working with the EDSAC computer (1948) involved *entering* and *leaving*

\(^{10}\) https://en.wikipedia.org/wiki/Maxwell%27s_demon
subroutines\textsuperscript{11}, as if subroutines were places (mazes?). though this is a workable metaphor today it was even better suited to a day when memory was physically large (room-sized).

the demon metaphor also allows us to blame our unseen inscrutable demons for unexpected side effects, bugs, and other problems caused by our own mistakes. and we can use all the help we can get.

\textbf{DATA METAPHORS}

from the earliest days of electronic computing the dominant metaphor for computer memory has been \textit{space}, as in physical space. we speak of memory size, room, capacity. memory is where one \textit{puts things}. memory is storage. and the name obviously refers to human remembrance (which tells you something about our metaphors for our own cognition). things put into memory persist over time. computer memory is assumed to be infallible (and it generally is, today).

in most general-purpose ("desktop") computers the same memory device stores both programs and working data but Arduino and many other similar controllers are unique: they contain different (physical) types of memory for specialized purposes. programs are stored in non-volatile Flash memory while working data (variables) is stored in RAM\textsuperscript{12}. the contents of RAM persist only while power is on (called “volatile” memory). EEPROM memory is used for persistent storage but only for smallish items; it is too slow to write to and too expensive for general-purpose data. in general Arduino programs do not have access to Flash, and so all references to “memory” here refer to RAM unless stated otherwise, and since embedded controllers run only one, fixed program, and do not load other programs from some outside storage device these limitations are not a problem here.

\textbf{MEMORY ADDRESSING}

the logical arrangement of memory has it's own set of spacial metaphors. it is incumbent upon you to understand memory-organization metaphors or you'll never get anything interesting done. the concepts are not difficult. i will state the metaphors here as if they were physical fact. the arrangement of silicon and metal inside the chip may be otherwise, but from a “programmers point of view” the following are true:

\begin{itemize}
  \item memory is a linear array of identical cells (analogy: identical houses along a street).
  \item cells are containers (houses can contain any or no items).
  \item the \textit{address} of a cell determines nothing about the \textit{contents} of that cell.
\end{itemize}


\textsuperscript{12} Random Access Memory -- “RAM” is a sort of skeuomorph left over from the bad old days when main memory itself often involved physical motion to “access” memory cell contents, eg. access was sequential. in 2015 the last remnant of this are hard disks. in earlier days the main store itself could be a drum or disc, or, very early, acoustic pulses in a liquid or solid medium, or bright spots on a cathode ray tube (see Williams Tubes).
a memory *location* is “where” a memory cell is.

* a memory cell’s address is it’s location (strong analogy to “street address”)  
* cell locations are numbered from 0 up to some maximum (always a power of two).  
* cells can be written to, or read from; one speaks of “address X containing Y”.

Memory addresses are themselves data, eg. fungible symbols, generally thought of as numbers. since memory addresses are symbols, memory addresses may also be used as *data* -- a memory cell can contain a value that can be used as the address of any memory cell. though we might say that some memory location “is” an address, or a number, a character, a color, those are just human tendencies to think in terms of it’s use or application. in a crucial way, the contents of a memory cell is a collection of interchangeable symbols -- the meaning, if any, is in how it is used, not a property of the data itself.

**MEMORY CONTENTS**

*each cell/location* is itself an array of identical, independent *bits*. the bit is the smallest unit of storage in a computer, and can contain only one of two possible values, 0 or 1. Arduino memory cells contain eight bits. bits are also numbered; this number is sometimes called an address (“bit address”), 0 on the right, and 7 on the left. through no coincidence the bit-address number is also the exponent $x$ in the expression $2^x$, the decimal-weighted numeric value of that bit when you decide to interpret it as a number.
memory may be considered a one-dimensional array of cells (length only); each “unit” in the array is a cell. or memory can be considered as a two-dimensional array of bits, with dimensions being length (address) and width (bits).

It is a common fallacy that computer memory contains numbers. It does not, exactly; memory cells contain symbols. It is true that if you represent ordinary integers in base-two, that the resulting pattern of digits, 0 or 1, happen to “fit” directly into a memory location (metaphorically, at least), and this is not coincidence. But this does not mean that a cell full of bits is a number; that is simply wrong.
THINKING ABOUT PROGRAMS AND PROJECTS

how to visualize

theater metaphor

To know the world one must construct it.
-- CESARE PAVESE

the demon metaphor fits within a larger view of how we actually deploy our computers: theater. that the entity metaphor maps directly onto the role of an actor following a script seems obvious (and welcome) here. however, the context in which we use and embed controllers like Arduino is the most useful target of the theater metaphor.

embedded controllers are deployed differently than desktop computers. on a desktop one “uses the computer” rather explicitly, taking advantage of the generality available in a machine with many resources. we are usually consciously aware of using a computer. embedded controllers by definition are themselves usually invisible, hence the name. they exist to do something. a button is pressed and sound comes out; a knob is turned and color changes. when an embedded computer's task is to read the button or knob the computer itself is intended to be invisible. indeed, when this metaphorical mapping becomes visible (when it doesn't work correctly!) like an actor flubbing it's lines we become all too aware of the mediating machine.
media installations, especially, function as a theatrical performance when they are viewed as a system. so is the cockpit of a modern automobile, where the controls electronically mimic the mechanical controls of automobiles from the previous century. theater requires some seamlessness in the performance; we are there for the performance -- actors, script, props, stage, audience working together to produce the performance. that magical spell of a performance is easily broken; actors forget lines, props fail. too many of us have attended media installations where we wave our arms in the air asking “what does it do?” or have been frustrated when turning a knob labeled HEAT leaves us cold.

Programs are often called scripts for good reason. The word program itself derives from the Greek language, meaning 'public writing'. In theater a program describes to the audience what they are about to experience, and a script is a set of instructions the actors follow to create that experience.

**PROGRAMMING IS WRITING**

Computer programming is creative writing; formal writing. The languages are formalized and occasionally rigid, and it is certainly possible (and common) to write “correct” but terrible programs.

Just as it is unreasonable to assume that an author wrote a novel in one sitting beginning to end, so it
is unreasonable to expect the same from software writing. Writing is not a distinct act, it is a process that unfolds over time, and continuously evolves from your experience writing and testing that software program. You shall and must write multiple drafts. You will write throw-away test programs to tease out behaviors of components or techniques.

Luckily, unlike prose, the re-use of code is strongly encouraged -- cliche and self-plagiarism especially welcome. Once debugged, a working block of could should be reused as often as possible, and shared -- this is what open source means, and this is what software libraries are for. Cliches like using the same variable names in different programs is a really good idea when it means increased clarity and understanding.

Though there is pressure from industrial directions on rapid program development cycles, cost issues, etc, none of these have any bearing on the reality of writing good code. Good code takes time. If there is one universal truth to be found in the practice of writing software, it is this: it always takes longer. Always.

XXXXXXXXXXXXXXXXXXXXX INCOMPLETE
TECHNIQUE
TECHNIQUE

how to do it

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

-- GERALD WEINBERG

it is time to apply the ideas discussed so far to real program design and construction.

in an ideal world we would design programs from the top down, planning out what we want and where we want to go. in an ideal world, we'd have concrete goals to plan for! but even in industrial environments, which you are likely not in, we need to accommodate changes, challenges and opportunities found along the way. and when the goals are aesthetic our code needs to be technically open-ended. rigid rules for constructing real programs are therefore generally impractical.

at the detail level, when a program needs some thing to happen before it can proceed -- and of course this is routinely necessary -- the program must not block. blocking means to wait for something to change in a way that prevents the program from continuing. examples are for() or while() loops around a pushbutton switch, or the ubiquitous but ruinous delay() function.

at a structural level -- thinking about what a program does, and how it might change as you write it -- you need an approach to coding that lets you add, subtract and change large functional chunks without affecting previously-working chunks.

all of these are easily solved with a venerable technique: the state machine. this approach works on any hardware and in any programming language. in fact the ability to execute a state machine is the very definition of computer.

too much code is written as if it were for a 1960's batch job mainframe: get data, process data, output data. in the last half century computers were given input and output devices, called I/O; with it computers can now react to events and pass messages. today, a computer program that loops in place awaiting data or event is probably a poor program.

a more functional approach is to visualize the computer program running your program as reacting to stimulus. even if the program's only input is data and the only output is data, the receipt of data is an event that code reacts to, dispatched as quickly as possible to be ready for the next. no stimulus? no activity.

- physical push-button switch presses, sensor changes, any thing "seen" on the computer input pins, analog or digital, are events.
- any data arriving from any source asynchronously is an event.
asynchronous events arrive without notice or warning. synchronous events, those that arrive predictably (e.g. at a particular time), are also most easily treated as asynchronous.

for a program to react quickly to events, it needs to be ready. to be ready, it needs to not be stuck in a loop somewhere. only when your program isn't bogged down, usefully or not, can it react to an event. input pin).

an event is something that happens, and in a computer these will be electrical interface pins (digital or analog), the contents of some internal machine register (e.g. the clock), or the results of some computation, subroutine or variable (but input to those will originate at an interface or register).

events are changes in state. while anything with a detectable change can be an event we will concentrate on two sources: digital interface pins and the clock. with an understanding of these two you should be able to infer how to turn any change into an event.

events are of concern to us because they allow us to conserve effort and energy. events from the world outside the computer are relatively far apart in time. events require expending effort and energy; in between events we simply wait, time in which other things may unfold. events themselves are information.

a thing that is un-changing or static is said to be in steady-state. by tracking only change-events we (and code) can know the steady-state of the thing, if necessary.

HOW EVENTS ORIGINATE IN CODE

how to do it

embedded-control computers see physical world events indirectly through sensors, switches, etc attached to its input pins. it is at that interface that the state of hard matter is turned into data stored in a memory variable.

the CPU samples the state of some input interface at a moment in time. that sample then represents the state of the external event at that moment only. real-world changes “outside” the pin after that sample time go unnoticed until the input interface is again sampled. to be meaningful, for the sample to be an accurate representation of the outside world's state, the sample must be taken often enough so that any resulting action taken by the code in response to a change in state is perceived quickly enough to appear simultaneous. event data (information) within the computer pertaining an outside event is therefore never simultaneous; there is always a time difference.

all computer interfaces work in this way. a sound recorder samples input sound at a steady rate; video cameras sample an ever-changing image 30 times per second. there is rarely any advantage to sampling too much more often than necessary; already imperceptible reaction times can’t be
improved on, and collecting more data than necessary simply consumes resources to no practical advantage.

luckily for us, real world events that involve anything with mass, such as the human body, are very slow relative to the speed of any computer. events are generally measured in a range of milliseconds to seconds, or longer. it is widely accepted that even for demanding applications such as professional musical instruments, a response (lag) time of 10 milliseconds is imperceptible. few applications are that demanding. this is why most basic controllers don’t have resources built-in for high-speed data such as audio or video, which require responses in the nanosecond to microsecond range: low cost.

the above implies an ideological view of the nature of the world. due to their periodic nature, samples always represent the past; that the event/change that code detects lags the real event, and that very rapid changes may be missed -- because doing so allows us to use inexpensive, serially-sequential electronic machines to execute decisions and actions that we prepare for them. it is obviously a widespread ideology. we use it because we can reproduce results, not because in any sense it is “right”.

DETECTING CHANGE

let’s return to a physical analogy to illuminate the process of how we represent the concept of change and events in a computer: an ordinary household door. doors may be open, closed, or in any position between. we are not interested, here in this discussion, in in-between positions, and so we ignore those. we are only concerned when it is opened or closed. (if you think you detect an opportunity for an extensive discussion on the knowability of the state of physical matter, you are probably correct.)

your metaphorical task is to yell out whenever the door is opened. pretend you’re announcing guests at a party if you need a context/pretext. let’s unpack the obvious: how is it that you know that a door has recently been opened?

• look at the door’s current state (open or closed).
• if the current state is different from the previous one, yell out.
• repeat.

while writing this section, only upon reading what i had written did i notice that i overlooked a critical detail -- that what i meant to say was this:

• look at the door’s current state (open or closed).
• if the current state is different from the previous one, and if the door is currently open, yell out.
• repeat.

i had simply overlooked the important detail that it was the opening door that was important. it is all
too easy to overlook critical parts of things that are so ordinary in real life. i noticed my oversight only when i had written it and then later read it. the act of reading and writing, and not simply thinking, was required for clarity.

this sort of mistake/oversight is a common one that happens while writing anything at all, and is usually repaired by the author (eg. me) without the reader knowing. but i decided to expose this inner, hidden process here because it is precisely the sort of thing that occurs during the process of writing. i theory i could have/should have known to not make that particular mistake, and often i do; but many times a re-read is necessary, and more times, only later execution reveals the oversight when the script does not work!

computer programming is creative writing within a formal system. rarely will you get things correct the first time through. you will write your program many times, and you will revise it many times.
**TIME**

In the real world, you do not stare at a door continuously to know when it opens or closes. *We have other things to do*, and so does our code, as you will see. While you would certainly not miss a door opening event by endlessly staring at it, you would not have time to do anything else. Continuous monitoring is also simply unnecessary; doors take finite time to open or close. It turns out that few things in the physical world change very quickly, at least relative to the speed of even a slow computer, and so periodic checking is more than adequate.

The intuitive human solution is also the correct *software* technique -- glance at the door periodically, often enough to not miss changes, but not so frequently as to be a burden. But what is "often enough"?

The tiny sub-actions of looking, deciding, occasionally yelling but most times not -- each in itself takes almost no time to complete. In other words, most of the time you are *doing nothing but waiting*. Where a person might read a book or fall asleep, a non-metaphorical electronic computer can run countless other event-driven tasks in this manner.

This is the fundamental advantage of event-driven software: nearly all time is spent *waiting for events*. This is also true for actions, which we will examine in a later section -- most actions can be reduced to one or more near-zero-execution-time sub-tasks. This is the advantage of a cooperative event-driven strategy.

**TIME EVENTS**

*how to do it*

In some ways, time events are the most fundamental event in any computer program but especially so in event-driven real time programs. We treat time as an event in order to know when to perform a given action. We treat time as an event as part of a technique to *not block*. Code for timed events does almost exactly what we do ourselves; read the clock, add to that the desired time interval, remember it, and periodically check the clock for that time to arrive.

Time is a positive integer which begins at 0 when the computer is powered on, increasing by one every
1/100th of a second, up to a maximum of 4,294,967,295 (the maximum value an unsigned long int can hold). When maximum count is reached it overflows, the count starting over at 0 and increasing as before. 4,294,967,295 milliseconds is approximately 49 days.

The Arduino contains both millisecond and microsecond clocks; this discussion considers only the millisecond clock, accessed by the `millis()` library function which returns the current time as an unsigned long integer.

Calculating a time X milliseconds in the future is easy:

```c
unsigned long futureTime = millis() + X;
```

to treat futureTime as an event, simply compare to the current time:

```c
if (millis() >= futureTime) {
    // TIME HAS COME, DO SOMETHING
}
```

Extending this into a repeating timed event is also simple:

```c
if (millis() >= futureTime) {
    // is now event time
    futureTime = millis() + X;  // schedule NEXT event...
    // TIME HAS COME, DO SOMETHING
}
```

You might visualize this as hopping up the time line, X units at a time.

While these are perfectly correct and usable if you will be using more than one timer the hand-management of unique variables for each “timer” quickly becomes unwieldy. A practical solution calls for a bit more overhead and complexity, and a very practical timer library will be introduced later.

**SWITCH EVENTS**

> Everything should be made as simple as possible, but not simpler.
> -- ALBERT EINSTEIN

Possibly the most common and useful input is the electrical switch. Switches make or break an electrical circuit and come in many configurations, but overwhelmingly the most common type for computer interface is the single-pole, normally-open, momentary operation, pushbutton switch. See the XXXXXXXXX section for a more detailed description of how these are connected to the computer.

Recall from the metaphor discussion that detecting change requires remembering, and remembering is bound to time. Detecting a change in a switch requires code to manage three things: current switch state, previous switch state, and elapsed time.
with the arrangement given in section XXXXXXXX the `digitalRead()` function samples the pin's electrical state and returns it as a numeric value, 0 if closed/pressed otherwise 1. Given that in most instances we want to know when the switch was activated by some outside force (e.g., your finger) a common approach is something like this:

```java
if (digitalRead(7) == 0) { // BAD CODE
    // switch was pressed
    // DO SOMETHING HERE
}
```

This is almost always a mistake. It fails to detect the event of switch activation, instead responds to the mostly-unchanging rest or activated state of the switch. In fact, this code does the opposite of detect-change because it has no memory. This code is not of sufficient complexity to capture an event.

** XXXXXXXXXXXXXXXXXXXXXXXXXXXXX work up past, current, time event to sample. **
CASE STUDY: A SIMPLE SWITCH EVENT

code as theater; simple installation where entering a room causes a light to illuminate a static sculpture on a pedestal [after a delay].

CODE IS THEATER
THE SWITCH
DIGITAL SWITCH with PULL UP

digital pin 7

pulls up
10,000Ω
weak

+5V

pulls down
hard

GROUND

1
Ø

Ø

GROUND

THE EVENT!
ref door-open metaphor story here. repeated sampling of the door <-- switch <-- arduino pin as “looking”, aka reading.

the graph here is marked off in an in-human time scale -- tens of milliseconds. since doors do not move fast this is quicker than is needed.
isolated and abstracted, we consider the samples over time, each in a little container. we look for patterns -- and see a discontinuity in the sequence precisely at the time of the door/switch/pin event -- this is what we want to code for. this is the event.
we discern that we only need consider the current state of the door (switch, pin) and the previous state of the door. From that alone we can detect the change event. As in the metaphorical story the job is tedious but simple.

It's a good idea to play this out on paper for a few iterations to get the hang of it. Note the table of three instructions on the right, in orange. These describe the inner algorithm. "Inner" because this is like looking under the hood of a car -- sometimes you need to momentarily set aside the larger purpose of the task to concentrate on the details to make sure you understand them and that they actually do what you want.
of course we need to use physical containers, not thought-experiment containers, and our physical container is a byte location in the Arduino's memory. (it is quite physical, simply too small to see).
intro bit mask, i think here

1101
& 0011
-----
0001

BIT MASK

boolean arithmetic
etc., but more easily

---
1101
"BIT MASK"

---
0011

---
0001

some variable's contents

ROUGH DRAFT/PROTOTYPE   copyright Tom Jennings  tom@SensitiveResearch.com
complete the code frag
code for final event example:

```c
void setup () {
    Serial.begin (9600);
}
void loop () {
    if (switch_loop()) {
        Serial.print ("SWITCH PRESSED!");
    }
}

bool switch_loop () {
    static unsigned long T; // next scheduled sample time
    static byte S; // current and past switch samples
    if (millis() > T) { // if it's time to check
        T= millis() + 10; // (schedule next check, 10mS from now)
        S= S << 1; // NOW becomes PAST
        S= S | digitalRead (7); // set NOW to current switch state
        S= S & 00000011b; // mask off all but NOW and PAST

        // there are four possible combinations:
        // PAST NOW
        //  1  1 = 3 switch remains open
        //  1  0 = 2 switch recently closed EVENT <- THIS
        //  0  0 = 0 switch remains closed
        //  0  1 = 1 switch recently opened EVENT
        //
        // but here we only care about RECENTLY CLOSED.
        //
        if (S == 2) return true;
    }
    return false; // not time to check yet
}
```
loops are a powerful and fundamental concept in computing but also the most common cause of slow unresponsive code. loops are the most common way in which code blocks (for Arduino specifically delay() is worse). loops of all types enclose blocks of code that act on a range or sequence of data, often indexed with a loop variable.

CLOSED LOOPS

there are two kinds of loops, closed and open. closed loops are what most people think of when they hear the word "loop" -- block of code enclosed by one of a language's loop statement types; for(), while(), do(), etc. the statements so enclosed are executed repeatedly until the loop terminates.

closed loops are a form of subroutine, an "inline subroutine". the block of code within the loop is executed until loop termination. loop code is often a prime candidate for conversion to a full subroutine. closed loops are easy and satisfying. they they look and act like neat little self-contained subprograms. and that can be a problem -- when enough time is spent inside them, they block.

when using closed loops in an event strategy make sure that you consider all of the following:

- what causes the loop to terminate? loops terminate when some condition is met. if the condition is not met, then the loop will not terminate, and it will spend all of eternity grinding through the contained statements. some loops simply count, often indexing some datums (clearing an array, etc). loops with simple numeric counts often don't block unless the count is "very large" -- and that is dependent on what else is going on in the loop. filling an array with 0's is "fast"; calculating a Bitcoin block would be "slow".

- does the loop depend on time in any way? if a loop awaits for a future time through any means, such as delay(), waiting for a value to come up with millis(), then this is the very definition of a blocking loop, if loop termination depends on time in any way. this is covered in the TIME section and in the case studies. referencing time is perfectly fine; delaying or
a definite method: fastCode

awaiting a future time is blocking.

• does code within the loop wait for a particular state of any input? the thing about the unknown is, you don't know what it is. if loop termination depends on an input pin changing to some particular state, then that loop blocks, and worse, likely hangs your program in a stuck state\textsuperscript{13}. any change requiring human intervention or most external processes (weather, wind, light, whatever) needs to be handled as events, not in a loop. even when this sort of event is certain to happen, the time between events is again, almost by definition, too long to be put into a loop. it will block.

• how much time does it take for the loop to terminate, worst-case? while loop-execution time can easily be measured – refer to my SRLoopTally library – it's usually adequate to simply meet the guidelines above and not do anything that obviously blocks. without any concerted effort on my part, most of my modestly complex Arduino installations run 5000 loops per second, under 200 microseconds average, but with an occasional (every 10 seconds) loop that takes one or two milliseconds.

closed loops should be thought of as tools for solving quick self-contained tasks such as clearing a table, turning all LEDs on or off, rolling dice, calculating a value, all things that terminate almost immediately. closed loops must terminate deterministically -- closed loops cannot wait for any outside event to occur.

in general, for anything but the most trivial purposes, unwind all loops. Wiring's main loop() function is an unwound loop and discussed in the next section.

\textsuperscript{13} the (rare) exceptions to this are all classes of synchronous devices, such as complex integrated circuits or modules. these often require what's referred to as "handshake" (another useful metaphor), a form of protocol used to deliver data to, or retrieve data from, modules that are often themselves self-contained computers. hardware protocols such as SPI, 2wire, iButton etc are often involved. the non-blocking requirement still holds true; the data-exchange must still take place in a "non-blocking" amount of time, and the probability of "failure to complete" must be extremely low (eg. the chip hasn't failed). this sort of programming is outside the scope of this discussion, and i mainly include it to emphasize that a loop requiring that some particular input state manifest before the loop terminates is simply a very bad idea.
LOOP()

how to do it

loop() is not just a place to stick your code. a carefully structured loop() is the key to good strategy.

loop() is an unwound loop. the Arduino reference tells you that code placed in loop() executes from top to bottom and exits; after exiting loop() is immediately called again. this process repeats endlessly. this is a loop, unwound like tape off a spool.

it may help to think of loop() as containing a series of tasks to be executed one by one, to get from the first statement to the last in the shortest period of time. this is the overriding goal of this programming strategy.

code placed in loop() -- and all code called by your code, all the way to the bottom of every library function called -- must not block. all code called in loop() must complete/return as quickly as possible without waiting. how this is done is covered in a later subsection.

it is important to be aware of the path the CPU follows through your code as your program runs, and this will be examined as we proceed. as the demon executes your program, it follows the path of instructions, setting variables and changing outputs, reading variables, branching at conditionals, etc.

let's examine the main loop function from the FANCY BLINKER case study in Section XXXXXXXX.

```c
void loop () {
    blink_loop();   // blink the LED
    down_switch_loop();  // watch for switches
    up_switch_loop();    // watch for switches
    report_loop();      // periodically prints blink rate value
    SRL.tally();        // loop() statistics
}
```

loop() simply invokes each task subroutine in order. loop() doesn't "know" what each does (though carefully chosen names and written comments help inform readers) and that is the point -- loop() simply invokes each of the tasks. the task subroutines do the work.

recall that our overarching strategy is to react to events quickly so that response is seamlessly real time without lag or delay. for this to happen each task subroutine must not block. the next section illustrates how that is accomplished, but for the moment assume that each task subroutine above takes an insignificant amount of time.

but how much time does each task take? and what is "quick enough"? the first question is easily
answered empirically: the SRL.tally() function measures \texttt{loop()} execution time and prints out a summary at the rate specified by SRL.begin(). below is a capture from an Arduino Uno running the FANCY BLINKER program:

\texttt{#loop: 36674/s, 27uS avg, 264uS max}

let's go over this terse report in detail. this program executes 36,674 iterations of \texttt{loop()} per second, for an average of 27 microseconds each. in the measurement interval (10 seconds in this example) the longest single \texttt{loop()} execution time was 264 microseconds. therefore that 27 microsecond average was calculated over some 36,674 iterations in the 10 second period making the average quite representative, with the 264 microsecond worst-case not much of an outlier.

but why the variation at all? the reason is that the statements executed within each task subroutine depends on conditions internal to that subroutine. \texttt{loop()} is quite intentionally isolated from the internals of the task subroutines. \texttt{loop()} simply invokes them one by one, endlessly, without knowledge of what goes on within them.

with average loop execution time of 27 microseconds each task-loop-subroutine takes just over 5 microseconds to execute. response times under 10 milliseconds are undetectable and feel instantaneous, so this code clearly exceeds that requirement by a huge margin.

\texttt{loop()} runs quickly because each task-loop-subroutine it calls runs quickly. if any task-loop-subroutine took a significant amount of time to execute this strategy would fail. this strategy depends entirely on the way that task loops are structured and coded. the method is deceivingly quite simple and explained in the next section, though the explanation may not fully make sense until you've gone through the rest of the subsections that follow.

as author, the task-loop subroutine container allows you to think only about what each task must do and not be distracted by the concerns other tasks. as long as code within the task loop doesn't block it can be completely isolated, without affecting other task loops.

the short answer, expounded upon below, is that the code in task loops \textit{do not block}. if you recall from the METAPHOR section, blocking is standing in one place waiting for something you need, preventing anything else from happening. because CPU code execution is sequential, code that blocks prevents all other code from executing -- the machine is effectively halted. when a task loop needs some resource in order to continue, it terminates, remembering where it left off, so that when \texttt{loop()} again invokes it later, it can see if the resource is available. as always task code must handle events current and past, and time.
SUBROUTINES

*how to do it*

It is standard advice to put code into a subroutine when it is used more than once. This is correct but barely touches the surface of the powerful advantages of subroutines. Subroutines are containers that help you think at a high level of abstraction about events, time and actions. Subroutines literally extend the programming language. Subroutines surround and contain complexity. Subroutines can reduce complex logic using the technique of *early exit*. Subroutines enclose code and data private to that code. Subroutines make programs easier to think about, easier to write, easier to read and more reliable.

Subroutines are *sub-routines* -- smaller programs that your program calls for *what it does* without being concerned for *how it does it*. This is the container metaphor, full-blown. Subroutines synergize fully with how we think. Subroutines are how code becomes reusable and shareable -- all library code is contained in subroutines.

It may be possible to use too many subroutines, but I have never seen it. Each time you need to write new code, start by creating a new subroutine. It can't possibly hurt.

While subroutines can be written to accept any number of input parameters, they can have only one (or none) return value. The reason for this is partially historic, due to the original intent of the C language as a "portable assembler". It is a constraint that can be used to advantage to structure your code. A subroutine invocation can be inserted into any control-flow or assignment statement and tested as if it were a built-in, a library function or a variable.

Here's a simple example.

```c
if ((number < 1) && (number > 100)) {
    Serial.println ("number is out of range");
}
```

Here is the same code subroutined. There is no real difference in execution but readability is increased:

```c
if (OutOfRange (number, 1, 100)) {
}
```

---

14 An assembler converts a source (program text) file and renders it to a binary object file for direct machine execution, much like the C++ compiler, gcc, does except that the source statements are the target CPU's instruction set, and there is a one-to-one correspondence between statement and machine instruction. Writing in assembly language gave way to "high level" compilers some time back in the 1960's. Writing in assembler is today considered macho (sometimes pointlessly so) though the occasional bit of strategically placed assembly in very specialized applications is occasionally warranted. I was an assembly language geek in the past but I haven't written any in decades, this cooperative strategy replacing nearly all instances of the "need" for assembly in interrupt service routines, by obviating the need for hardware interrupts in the first place.

ROUGH DRAFT/PROTOTYPE copyright Tom Jennings  tom@SensitiveResearch.com
Serial.print ("number is out of range");
}

// return true if the input number is outside of the given lower and upper bounds.
//
bool outOfRange (int number, int lower, int upper) {
    return (number < 1) && (number > 100);
}

why bother with the extra effort? one reason is that programs expand as we write and use them. programs are evolving creative works. outOfRange() is now a general purpose function that can be used elsewhere in the program and in other programs. good subroutine names suggest their function, and so the resulting code reads more clearly. if you need to change your code you only have to make one edit if that code is contained in a subroutine.

read aloud, the statement “if out of range, print message” though a bit terse reads a bit more like prose. most importantly, subroutines hint at a way to think about code, a strategy for coding effectively, to push complexity into manageable containers that you can make work one by one to build a larger program.

this is one of those subjects where the rules can only be expressed as a collection of do-and-don't anecdotes because subroutines are so flexible that an exhaustive study isn't possible or practical. if you put related program statements together into a subroutine you will stumble upon most of the advantages if you pay attention and remember some general rules of thumb:

• when you think of a section of code as “doing some thing”, subroutine it.
• when you have code that calculates a value based upon inputs and a bunch of control-flow statements, subroutine it.
• bounding is one of those things that comes up a lot in good code: determining if a number is within a range, or forcing an value to be within a range (within bounds). these are wordy and messy when done in-line; they are clean neat and readable when subrouted.
• printing messages for humans is a messy illogical task because human languages are messy and illogical. subroutine it.
• if you need to do something if and only if a large number of conditions are met by all means subroutine it.

i have an Arduino project that uses a purchased add-on board that stores sound files, and plays those sounds upon software commands issued according to a timed schedule. the code for initiating playback is complex, depending on a number of settings and possible errors of the board. to play sound file N the code must:

• when it is the correct time to play (time event)
a definite method: fastCode

- be enabled (not “muted”)
- sound track N must exist

all of the above conditions must be true to play a sound; and if not, each of them must generate an error message. put inline it becomes a messy tangle of multiple if..else if..else if.. conditionals. but put into a subroutine, it becomes easy to read and maintain:

```c
void playSound_loop () { 
  if (! timer (PLAYTIMER)) return; // return if not time yet
  if (! enabled) {  // test for MUTED
    Serial.println (“NOT ENABLED”);  // not error per se but report
    return;
  }
  if (loadTrackNumber (n) == 0) {  // try to select the track...
    Serial.print (“FILE DOESNT EXIST”);  // oopsie
    return;
  }
  playSound();  // initiate sound playing
}
```

most important of all is that the entire task is done with a single subroutine call with no additional intervention. the above code is of course a very small, abstracted and simplified version of the real code. but in the actual project the task of sound playing is handled exactly the same. loop() invokes it simply with:

```c
void loop () {
  playSound_loop();
}
```
SEQUENCE

how to do it

How often I found where I should be going only by setting out for somewhere else.

-- R. BUCKMINSTER FULLER

computers are sequential machines. statements execute top to bottom until control flow statements change the sequence, based upon some condition. program execution follows the path metaphor, with some non-human extras added (metaphors have their limits).

though our focus will soon be with the nuts and bolts of the programming language, it is important that during the coding process that you do not lose track of your intent for the program. how do we break a given task into smaller, simpler, "atomic" chunks? how do we arrange for those chunks to get done in the right order? when the task is extremely simple, such as the canonical BLINK example, the sequence of actions is obvious enough. you can probably puzzle out from examples and the Reference how to skip (make conditional) small blocks of code. the larger problem is that strategic ways and means, tools and metaphors for escaping simple linearity are not located in programming language references, because they are higher-level human constructs.

you do not code your way out of needing to block, you plan and strategize to not need to before you write a single line of code. non-blocking code doesn't await anything in a loop, or with the horrible delay() function. non-blocking code exists in unwound loops and state machines.

to react as fast as possible to events in your machine, when a resource needed isn't available, you perform other tasks while you await the resource.

sequence refers to the order of execution of program statements. statements are executed in the order in which they occur. execution order is affected by control-flow statements, but also by subroutine call and return.

program control flow is categorized as one of two types:

• trivial sequences are simply inline default order, or with conditionally-skipped blocks eg. if(). trivial sequences are always forward, and never branch backward or loop.
• a complex sequence is any sequence that executes in any order other than top to bottom, eg. branch backwards, loop, or more than “a few” conditional blocks.

TRIVIAL SEQUENCES

Arduino's loop() is a trivial sequence. trivial refers exclusively to the order of execution of statements,
not what the task loops might do. the subroutines in loop() are executed in order from top to bottom without variation and loop() repeats endlessly.

all of the task loops in the FANCY BLINKER case study are trivial sequences; for the most part, once the time condition is met the code simply executes in-line, with the switch tasks having a single if() statement that inherently does not block.

simple loops with deterministic terminations and are inherently quick -- clearing memory, turning all LEDs on or off -- are often trivial sequences and can exist within closed loops without problems.

**COMPLEX SEQUENCES**

complex sequences are where program complexity (and bugs) often explodes when constructed with ad hoc logic, such as multiply nested if..else statements. nested if..else sequences quickly get out of hand; as little as two-deep and the resulting logic becomes impenetrable, inflexible and unreadable. nested if..else logic is very difficult to modify.

most interesting interactive things we want to do with a computer are complex sequences.

the solution to constructing any sequence more complex than the most trivial is the *state machine.*
THE STATE MACHINE

how to do it

"Contrariwise", continued Tweedledee, "If it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

-- LEWIS CARROLL

state machine programming may seem alien at first, but they are incredibly simple. do not let mere unfamiliarity get in the way of mastering them, because once you do you will have a way to think about tasks of any complexity, and you will wonder, as i do, why they are not universally used. state machines are capable of handling any sequence possible to be executed by a computer. there is much theoretical treatment of state machines on the internet and in the literature and i swear, seems intended to obfuscate and confuse, because few things of such power are easier than a state machine.

a state machine is an abstraction that can be constructed in any programming language using ordinary control-flow statements, eg. if(). the C language however has a control-flow statement specifically to build state machines: switch..case.

the switch..case statement is a metaphor for a multi-position rotary electrical switch. instead of only two positions, on vs. off, a rotary switch can be turned to one of N positions. they are very common on appliances to select amongst modes of operation, etc. to construct a state machine from the switch..case statement we need to add a state variable, an integer that represents the position of the switch's shaft that selects one-of-N available positions.

as our first example let's construct a state machine to blink an LED. the blink_loop() task subroutine in the FANCY BLINKER case study will be simpler than what we construct here, but the state machine
method has advantages that will become apparent later.

the order of decisions and actions in blink_loop() are trivial; once the scheduled time is reached the rest of the statements are executed sequentially. the state machine makes the sequence explicit and controllable, even though in this instance it is not necessary.

let's examine execution order in the FANCY BLINKER blink_loop() task. begin by mentally labelling the statements in blink_loop() 0, 1, 2, and 3. when it is not time yet, (millis() < T), the execution order is 0, 0, 0, 0, 0, 0, ... the demon does not get past the first statement. the moment that the blink time is reached, execution order is then 0 (true), 1, 2, 3 and the LED changes state. upon next iteration the sequence will again be 0, 0, 0, 0... since step 1 re-scheduled the next blink time.

in a state machine the order of execution is determined solely by the state variable. the contents of the state variable directly controls the position of the rotary switch. with it we can calculate statement execution order.

here is the blink_loop() code converted to a rather awkward state machine:

```c
void blink_loop() {

        static unsigned long T;
        static byte led;
        static int state;

        switch (state) {
            case 0:
                if (millis() >= T) state= 1;
                break;

            case 1:
                T= millis() + blinkPeriod();
                state= 2;
                break;

            case 2:
                led= ! led;
                state= 3;
                break;

            case 3:
                digitalWrite (LEDpin, led);
                state= 0;
                break;
            }
        }
```

let's review the topology of the switch..case statement, which begins with switch (state) and encompasses the block of code between { braces }. each case has a matching break and case..break
encloses a block of code. switch..case..break defines a rotary switch, with each of the case..break blocks a switch position. state is the ingeniously named state variable.

**blink_loop()** is invoked repeatedly from loop() as as before. the first statement evaluates the state variable. the state variable answers the question “where did i leave off?” or “where was i?” as you please. this is loosely analogous to an awareness of task -- a intentional rememberance and choice of where to continue. though in this example the sequence is trivial this ability to control execution order shows the power of a state machine.

given that initially state is 0 the switch statement passes control to the first case, 0. statements are executed until the break is reached, where the switch statement terminates, control passes to the statement following the closing brace and the subroutine exits.

the statement in case 0 only checks if sufficient time has passed. if not the condition is false, state does not change, the switch terminates and the subroutine exits. this repeats until blink time arrives -- only then is state= 1 executed. therefore the order of statement execution is the same as before; 0, 0, 0, 0, 0, 0, 0...

once blink time has arrived the condition in state 0 becomes true and state= 1 is executed, the break is encountered, the switch terminates and the subroutine exits. it is not until the next iteration that the switch statement passes control to case 1, which then schedules the next blink time, sets state to 2, exists the switch statement and the subroutine returns. execution order is then 0, 0, 0 ... 0, 0, 1.

this process continues with for the other states and the statements perform the LED-blink operations as before. finally in state 3 the LED is changed, after which state is again set to 0, beginning the process anew.

this initial example state machine has some silly features due to the overly literal way that i converted it from inline code for illustration purposes here and we will fix those next. but some comments are warranted before we do that. though this appears to be a lot of code statements (“verbose”) in each call of the **blink_loop()** subroutine very few of them are executed each time. each pass through the code is very quick. and though there are far too many steps -- we'll reduce it to two steps, next -- each is cleanly explicit. it is for this reason that state machines scale well. they may seem too complex for simple tasks, but they scale well. each state contained between case..break is itself like a little sub-program, with very easy to read logic as what is required to change to the next state. sequences of any complexity can be created cleanly and easily. clarity is far more important than brevity in this example.

it is a simple matter to insert a new step between two existing states of a state machine. this is often impossible to do in nested if..else code.

let's turn the above example into a succinct state machine that we can expand into something more
interesting.

the only thing that requires waiting is the blink ON or OFF time. the rest of the statements are straightforward and can be combined into one state. the first state (case 0) awaits the time event as before; the other schedules the next time event, changes the LED to the opposite state and changes the state back to 0.

void blink_loop() {
static unsigned long T;
static byte led;
static int state;

switch (state) {
  case 0:
    if (millis() >= T) state= 1;
    break;
  case 1:
    T= millis() + blinkPeriod();
    led= ! led;
    digitalWrite (LEDpin, led);
    state= 0;
    break;
}
}

just as in the handling of events, the present, the past, and time are represented. the state machine and the state variable make up a rememberance of what has already been done (previously executed case/state) and what must be done now (current case/state). the passage of time, as a series of steps, is made explicit.

the evolution of this code example, from four very simple states into two more compact states is also intended to be an example of how to think about coding your own solutions, and a concrete illustration of thinking-on-paper (even virtual paper). concentrate on clarity and correctness and readability. succinctness and brevity, used appropriately, come with experience and virtuosity.
BLOCKING

*how to do it*

now that we have introduced all of the basic techniques it is time to reexamine the critical feature of this strategy, the avoidance of blocking. here a truism is useful:

*code is fast, the world is slow.*

it should be clear by now that awaiting any resource or event by looping in place blocks execution and must be avoided at all times; and that the solution to this problem always involves knowledge of the past, the present, and time.

the particular structure of `loop()` with it’s trivial sequence of task-loops, and task-loops as state machines that do things based on a state variable, brings with it many advantages not yet outlined here. one of those is *testability*.

the accepted technique for debugging Arduino programs is very primitive, and is much as things were in the 1960's. usually all we have at our disposal to reveal bad logic and mistakes is embedded print statements, LED blinking, etc. therefore effective debugging requires careful thought and disciplined strategy.

state machines help this process incredibly. with them you can code a state specifically to print out debug info at a human rate, or when a particular condition occurs. thinking in terms of state machines

*render task loops as tifs and paint control flow over it in color -- one color for each path will allow one image to suffice for multiple states.*
PROGRAM LAYOUT

*how to do it*

once your design is scrawled out, program layout can begin.

the Wiring default layout is as good as any and better than most. my programs tend to use the same overall layout regardless of function or complexity. keep in mind that the source file is a script -- a set of instructions to the machine and a description of what you the author intend. here's a very rough overview:

i follow this as a general guideline and violate it whenever appropriate. the idea here and throughout the practice of writing code is to use repetition and familiarity to our advantage. when i need to find a variable, a subroutine, or a declaration, i know approximately where to look without searching. given the large amount of trivia a routine structure helps. this is one way in which code-writing fundamentally differs from prose -- creativity is expressed not at the level of word-craft but in larger functional organization.

strive to include background and supporting information in comments. i tend towards logorrhea in comments, and often have more comment than code. comments are my intent; code is the implementation. code is not self-documenting -- code does not contain *meaning*.

where possible and not excessively excessive, i try to include copious hardware documentation in the comments for pin declarations, eg. part and pin number, wire colors, that sort of thing. often when debugging you just need a hint to identify a thing to test without referring to schematics etc.
DATA DESIGN

do a “lets examine...” for setup() and loop() and task_loop()

VARIABLES

it is a common fallacy that computer memory contains numbers. it does not, exactly; memory cells contain symbols. it is true that if you represent ordinary integers in base-two, that the resulting pattern of digits, 0 or 1, happen to “fit” directly into a memory location (metaphorically, at least), and this is not coincidence. but this does not mean that a cell full of bits is a number; that is simply wrong.

the CPU has particular built-in instructions to manipulate memory contents in all sorts of interesting arithmetical and decidedly non-arithmetical ways. these manipulations are most deep fundamental tricks for doing fun, useful things with a computers. if you imagine each bit to be a physical block or square (as turing did) then with machine instructions you can perform “tricks” with cell contents (“bits”) as you can with simple physical manipulation of blocks.

NEED BIT SHIFT ROTATE ETC GRAPHICS HERE

you are limited here only by your ability to visualize “tricks” for sliding, flipping, rotating, copying, transposing. the CASE STUDIES illustrate some of the things that you can do with what is called bit fiddling.