Understanding Flow Charts

INTRO

The use of Flow Charts has been around for a long time as a means of developing a strategic plan or algorithm for solving numerous kinds of problems. Flow charts have been used in everything from business, industrial processes, computer programming, and even baking a cake. While numerous other methods have been devised over the years flow charting remains nevertheless a simple yet effective way of solving many problems in computing. In this article, I discuss the most common elements and adapt their usage to reflect typical situations encountered in computer programming.

It should be noted that I do not always strictly follow tradition in my terminology or usage of the symbols that will be introduced. In part this is because many of the traditional symbols were developed in the day of card readers and reel to reel computer tapes and have long since lost their relevance. I have therefore exercised some liberty in readapting their original role to something a little more applicable to present needs.

What is important is that the reader appreciates that flow charting in general is a way to force us to think through problems in a logical and orderly fashion. A well written flow chart can almost always be ported over to assembly language, or any other computing language, often in a near one to one fashion. This makes flow charting especially powerful as a way of communicating an algorithm across computing platforms.

START & TERMINATORS



To mark the beginning and end of each Flow Chart is the basic oval symbol depicted above. The start of the Flow Chart (point at which program execution begins) employs a header in which is written the name of the program, subroutine, or overall process. Sometimes people simply write "START" or "BEGINNING" for their entry point but it is more meaningful to identify the process or subroutine name; something that identifies what the overall Flow Chart is about in a few words. This is especially important if the flow chart does not explicitly document the overall objective elsewhere, like in a drawing title. The start symbol may be referred to as a header but its shape and basic purpose are fairly universal otherwise.

Similarly, the end of the Flow Chart must be identified, again usually with another labeled oval. Here it is customary to use the word "END" or something similar in meaning. In my example above, I use the assembly instruction "RETURN" to indicate that this is where my subroutine ends and returns to the main program. For the main program itself, "END" might be the better descriptor to differentiate it from a subroutine. In either case, the purpose should be obvious to the reader.

Some authors differentiate between the start symbol and end terminator by using more circular shapes for the start and restricting the oval depicted above to the terminator. The difference has not proven important in my case and I use the one shape for both. The labeling and position of my symbols is what makes the difference in purpose clear. Also, because subroutines in assembly instruction end with the "return" instruction, I frequently use a "RETURN" terminator and sometimes more than one which reflects how my program is actually constructed. If I can direct the flow chart's end to one common "RETURN" I will, as long as it doesn't make my flow chart messy with lines running everywhere and over top of each other. I like my flow charts to be pretty.

A subroutine that is called from another part of the program does not care which *return* it encounters first in order to terminate the routine (task), any *return* will do. Some may be disturbed to see more than one *return* used within a program subroutine and are inclined therefore to park a single *return* at the end of their subroutine code with multiple "goto's" planted within the routine directing all exit traffic to that one *return*. This is unnecessary however and only wastes another processor clock cycle to implement. Many programming purists are loathed to use goto's in any case, arguing that they lead to unstructured programming and risk jumps to regions outside the subroutine boundary. If your code reaches a natural exit point at multiple positions within the code, put a *return* there, it is a perfectly logical thing to do. There are many roads leading out of a city; you don't always have to take the same road when leaving. Still, when it comes to the flow chart, some will insist for their own reasons on only ever seeing one terminator.

As far as the start is concerned, here I must insist on one start point which in assembly language is identified by a unique label (e.g. "Subroutine1"). So while it may be equally true that there are many roads leading into a city, when it comes to program subroutines, all the roads leading out are one ways, and there's only one leading in.

DECISION MAKING



Another commonly used Flow Chart symbol is the diamond shaped *decision* symbol. In this case a condition is evaluated to see whether it is true or false ("YES" or "NO"). For example: "Is X greater than Y?" A simple binary outcome then results with one of two possible paths being executed following the decision block, depending on the result of the conditional test.

It is possible to implement a decision making task with more than just two possible outcomes, though by far the most common form is one that results in either a "YES" or "NO" response. An example of a multi-path decision block might be one which asks the question "what range does variable 'X' fall within" in which case the possible ranges could be 0-25%, 26-50%, 51-75%, and 76-100% which would result in the process taking one of four possible paths. Ordinarily however more complex decisions such as this can ultimately be broken down into simpler Yes/No questions which are always easier to implement in assembly code.

For example, we could have broken the range problem into successive questions like: "Is X<26%?". If false, then we'd ask the next obvious question: "Is X<51%?", and so on until we cover the entire range, each time creating a decision path corresponding to the answer to each test, as illustrated below. Note the inclusion of arrows which show the direction of program flow. These are an essential component of Flow Charts and only ever assume one direction (i.e. program flow can only proceed in one direction along any given path).



TASKS (PROCESS SYMBOL)

Every program ultimately requires that some intermediate task be performed, whether to calculate some math function, move data, or whatever. To represent tasks that do not explicitly act on physical inputs or outputs, the more generic symbol is simply a rectangle as shown below.



I use this symbol whenever I am performing some math function, copying or moving register values, or modifying bit values of internal registers (e.g. setting a software flag). This symbol can generally be thought of as performing some *action* or *task*. It is also referred to as a "process" symbol. The term task however seems to be gaining more traction in computing lingo these days but to each his own.

SUBROUTINE



Similar to the *task* or *process* symbol is the *subroutine* symbol drawn above. This rectangle with two double sides is sometimes referred to as a "*predefined process*" symbol (i.e. a process that was previously defined). In effect, it represents a more complex set of procedures, sub tasks, or entire other program. This symbol is useful when you're developing code for a main program that serves as a task manager and each of the many sub tasks can then be represented by this symbol. A separate flow chart would then be used for each sub task (subroutine) that details their operation.

PAGE CONNECTORS



When a flow chart cannot be fully constructed on one page, page connectors are used to show that program flow continues on another drawing page (sheet). The direction of flow is indicated by sender and receiver symbols. I like to draw my *sender* symbol using "to" in the label with the pointy end opposite the line connector. Similarly, the connector where program execution continues (i.e. the *receiver*) is drawn with the connector tied to the pointy end of the symbol like a tag. The *receiver* input is labeled as "from" to show that the flow chart's program flow continues here *from* somewhere else (the source location should be identified in the label). The *sender* and *receiver* symbols may be called *input* and *output* respectively or some other similar idea. Some drawings use grids that further aid in identifying the *sender* and *receiver* locations on the drawing, especially where multiple off page connectors are used.

The circular connectors are used where flow chart connector lines cannot be easily drawn from point A to point B within the same drawing and must be broken. In that case, these connectors show that a line continues at a different point on the same drawing. Some labeling system, like the aforementioned grid lines, must then exist to aid the reader in locating the start and finish points of a broken connection to be able to follow the logic flow. Alternatively, the flowchart could simply use unique alphanumeric labels like "A", "B", "C", "1", "2", "3", etc. at both line ends.

If multiple lines must be connected on the same page using page connectors, then each would share the same label, similar to multiple occurrences of the "ground" or "common" symbol on an electrical schematic. Just to emphasize the point, all lines that represent the same point (node) must be labeled the same and arrows should also be used to clearly demonstrate the direction of logic flow along lines.

In electrical drawings, page connectors are often just arrows or symbols similar to the off page connectors above. The point is to use symbols and labels in such a way as to clearly indicate the direction of the flow chart's logic; there should be no ambiguity as to what is meant in the drawing.

MISCELLANEOUS SYMBOLS



Additional symbols used in flow charting are those depicted above and these may be used as alternatives to the basic rectangular *task* or *process* symbol previously introduced. These too were developed around older technologies and may or may not be applicable to the user. The *display* symbol for example was developed when CRT (Cathode Ray Tubes) were in vogue. I have adapted it in my example for an LCD display. I could just as easily have written the same instruction into a rectangular *process* symbol or the unique *output* symbol I have created below.

The *manual process* symbol was probably not intended for software but here I've exploited it to "energize a solenoid" (I figured a solenoid was electro-mechanical so it's about as close to a manual process as I could figure for software control, or maybe not). The *manual input* symbol is easier to see an application for. In this instance, I've shown an example for checking the numeric keypad for manual entry. The "card" symbol goes back to those days when a computer program was punched on paper cards that were then fed into a card reader. In my example, I've adapted it to read in an analog input.

The parallelogram symbol (leaning rectangle) is the *data* symbol. How this symbol was originally intended to be used is unclear to me but here I've demonstrated its use in copying data from one register to another. Additional symbols to these exist that relate to the kind of hardware that existed from the earliest days of computing through to the 1990's but have now largely become obsolete. There is no particular reason why a given industry cannot modernize the symbols to relate to its own peculiarities today and many in fact have. The two symbols below for example represent an old and new symbol applied for tasks that involve inputs and outputs, primarily external I/O as opposed to data moves or tasks inside the computing process.



The decision whether to use these or any newly created symbols in lieu of a simple rectangle for all tasks depends on the degree to which the distinction is deemed useful. The case may be drawn for at least using a unique symbol for external inputs and outputs, as I've illustrated above, to differentiate from tasks performed strictly within software. This is because reading and writing to I/O transcends the software environment and crosses over into the real world. Using these additional symbols therefore permits the reader to more readily note those tasks visually that engage physical hardware.

The output symbol could be used generically for any and all forms of output devices, whether solenoids, relays, motors, other microcontrollers, or display elements like LCDs, LEDs, or serial communications.

The argument could be made for further subdividing these into more explicit symbol types but too many symbols does not necessarily improve the point of the flow chart and may in fact make it more difficult to interpret, not to mention draw. The symbols that I have described thus far have served me well and I have not found any burning need for more.

Again, the importance of the flow chart is to aid the designer in breaking the problem down into a sequence of smaller logical steps (tasks and decisions) that ultimately lead to a workable solution. Of course, if you're doing it to win a drafting contest, then knock yourself out.

An example of a flow chart for implementing a motor overload with i^2t trip function is given above. In this example, the motor amps are read in another routine and stored in the 8 bit binary word ADRESH (Analog to



Digital Result High). OLPU is the overload pickup target and OLSum is a 16 bit overload timer that accumulates the i^2 t product (this represents the thermal energy in the motor proportional to the product of amps squared and time). The flow chart was written for conversion to assembly code using Microchip's PIC18F4520 MCU (Microcontroller Unit) as a simple demonstration of how modern current overload devices use embedded solutions in lieu of older electro-mechanical thermal devices. Note that program flow is explicitly shown throughout the flow chart using arrows.