Choosing the right microcontroller: A comparison of 8-bit Atmel, Microchip and Freescale MCUs

Mel Slade, Mark H. Jones and Jonathan Scott
Faculty of Engineering, The University of Waikato
November, 2011

Abstract—**When choosing a microcontroller there are many options, so which platform should you choose? There is little independent information available to help engineers decide which platform might best suit their needs and most designers tend to stick with the brand with which they are familiar. This is a difficult question to answer without bias if the people conducting the evaluations have had previous experience with MCU programming predominantly on one platform. This article draws on a case study. We built three "Smart" Sprinkler Taps, small, self-contained irrigation controllers, differing only in the microcontroller unit (MCU) on the inside. We compare cost, development software quality and hardware performance from the perspective of a new user to each of the platforms.**

I. Introduction

This article sets out to test three popular families of 8-bit microcontrollers, Microchip, Freescale and Atmel, by comparing MCUs each having similar specifications (as illustrated in table I). The selection criteria was based on needing a low power MCU with a minimum of 24 general purpose pins, an ADC and timers for only a few dollars.

	Microchip **PIC16LF1938**	Freescale **MC9S08QB8**	Atmel **ATtiny88-AU**
Cost (NZD)	$4.39	$1.98	$4.37
Pin Count (I/0)	25	24	24
Frequency (MHz)	32	20	12
Frequency (MIPS)	8	20	12
Flash (Kb)	28	8	8
EEPROM (bytes)	256	N/A	64
Ram (bytes)	1024	512	512
ADC Channels/ Max Resolution	11 10-bit	8 12-bit	6 10-bit
Timers	5	3	2
PWM	Yes	Yes	Yes
Communication	SPI/I^{2}C	SPI	SPI/I^{2}C
Internal Temp Sensor	No	Yes	Yes
Comparator	2	1	1
Supply voltage (V)	1.8 - 3.6	1.8 - 3.6	1.8 - 5.5
Standby Current (nA)	60	250	100
Operating Current (μA)	150 @ 1MIPS, 1.8V	750 @ 1MIPS, 3V	243 @ 1MIPS, 1.8V

TABLE I: Specification comparison of the microcontrollers

The task chosen to serve as the case study is the development of a "smart" sprinkler. This was done for each of the platforms in parallel by three final-year undergraduate electronic engineering students. The task involved taking an existing product (a Garden Mate® two dial automatic tap timer) and adding various extra facilities. The units were to synchronise themselves with dawn, spread waterings throughout the daylight hours, log activity to non-volatile memory for later analysis, and most importantly alter watering patterns, after a calibration period, based on changes in environmental stimuli measured by the processors. These stimuli included temperature and ambient light via the addition of a temperature sensor and photo-transistor. Provision for future incorporation of rain sensing was also included.

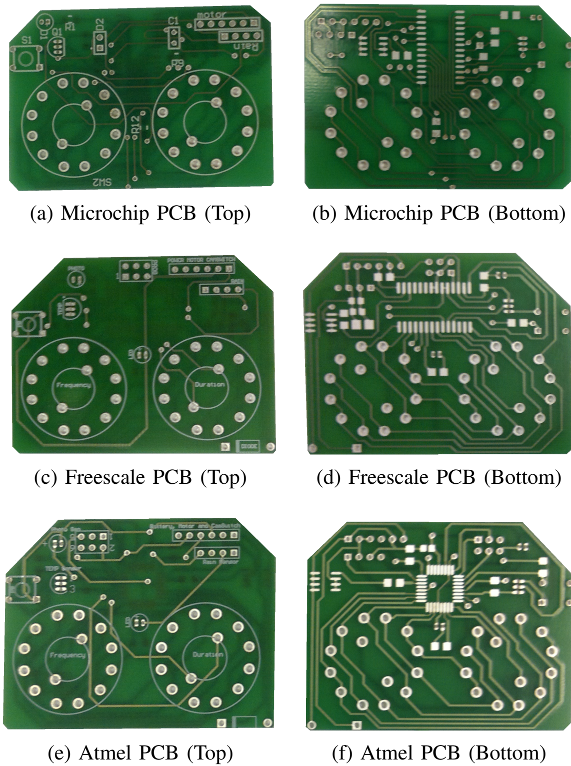
Unified Modelling Language (UML) activity diagrams were used to design the code in order to keep the programs consistent across platforms. The use of a common set of documentation diagrams ensured that the software will behave the same across each of the platforms while providing a convenient way to document the software operation. The diagrams used for this project are included as an appendix of this document, as are the operating instructions that might accompany the tap timers.

The modified tap-timer allows the operator to select the frequency and period of watering cycles desired throughout the day. For the first week, the system will adhere to the user's input, by opening and closing as often and for as long as set, whilst gathering baseline temperature and ambient light data. Once this calibration stage is complete the tap-timer engages its "smart" mode where it adjusts the duration of each watering window to match the previous day's conditions.

II. Hardware

The original tap-timers were disassembled and retrofitted with circuit boards designed around each of the new microprocessor footprints. Figure 1 shows both top and bottom views of each of the replacement boards. Apart from the microcontroller, programming header and connection for an external temperature sensor (on the Microchip board), the circuits are laid out similarly. Figure 2 illustrates that from the outside, the three sprinklers appear identical. The tap-timers are fitted with a pair of 1.5V alkaline batteries which are also used as the power supply for the new designs. The same input switch locations and mechanisms, LED position and push button are employed to enable minimal modification to the case. An aperture is added to admit light for measurement of insolation and detection of dawn, for solar synchronisation of the watering cycle.

The Atmel and Microchip microprocessors are similar in



(a) Microchip PCB (Top) (b) Microchip PCB (Bottom)

(c) Freescale PCB (Top) (d) Freescale PCB (Bottom)

(e) Atmel PCB (Top) (f) Atmel PCB (Bottom)

Fig. 1: Top and Bottom Views of the Three “Smart” Sprinkler Platforms Illustrating How Similar They Are.

Fig. 2: The three sprinkler prototypes which are outwardly identical, with a different micro-controller in each.

cost although the Microchip MCU boasts much larger an amount of program space, four times the amount of EEPROM and lower power consumption figures. The Atmel has a higher instruction frequency and wider operating range but for the battery operated smart-tap-timer, low power, extra code space and increased non-volatile memory are considered more important. It should be noted that the Microchip range of MCUs have an instruction clock that is fixed at 1/4 of the oscillator frequency. This means that a Microchip MCU with a clock frequency of 32 MHz (such as the PIC16LF1938) equates to an instruction frequency of 8 MHz or 8 Mips.

The Freescale chip is less than half the price of both the Atmel and Microchip MCUs, however a major shortcoming is its lack of EEPROM. Its flash memory can be written in software but as the MCU is based on Von Neumann architecture there is no distinction between program memory storage and user storage space, so it is a case of user beware.

III. Development

Despite the fact that the three micro-controllers seem fairly equal, the experience with each platform and accompanying IDE was anything but equal.

A. IDE and Development Kit

Microchip and Atmel sell their programmers and supply their respective development environments (IDEs) free of charge. The Microchip PICkit 3 programmer retails for $68.00 where the equivalent Atmel AVRISP2 retails for $73.50. Freescale bundle their IDE and programmer (USB MULTILINK BDME) together which retails for $217.76.

1) Microchip: MPLAB v8 was used with the lite version (free) of the HI-TECH C compiler. MPLAB has a simple interface with easy access to most of the features required for compiling and downloading projects. MPLAB consumes very little system resources and had no noticeable bugs. On compilation of a project the compiler shows how much memory has been used as a percentage of total space available on the specific MCU. This was helpful throughout development as it allows the programmer to develop a feel for the memory cost of new code additions.

2) Freescale: The Freescale development kit included CodeWarrior, which is the Freescale IDE. CodeWarrior was crowded with a bloated feel and didn’t always run smoothly. This meant taking extra time to get familiar with what was and wasn’t needed. One redeeming factor for CodeWarrior was the highlighting of register names and global variables, allowing the programmer to spot syntax errors quickly. CodeWarrior supplies a graphical user interface (GUI) for generating initialisation code, which was found to be useful as this meant less time was spent referring to the datasheet. The IDE also featured a one-click compile and download button that, provided no errors were detected, could compile the code and program the MCU.

The freescale was the easiest to setup for programming as the datasheets for both the MCU and the programmer itself had the pin connections for in-circuit programming documented.

3) Atmel: AVR Studio 4 was used to develop code for the Atmel MCU, which uses the GNU C compiler (GCC) that is open source and widely available but had to be downloaded separately. Much like CodeWarrior, AVR studio had a heavy bloated feel. It should be noted that Atmel has a more recent version of AVR Studio (AVR Studio 5) which is based on the open source IDE Eclipse. This version doesn’t require the compiler to be downloaded separately, supports code completion and other advanced editing features not present in either MPLAB or CodeWarrior. The Atmel situation is changing, probably for the better.

B. Hardware configuration

The hardware configuration was largely dictated by the board layout and the UML diagrams so there was little freedom when deciding how to configure each of the MCUs.

The GUI provided with CodeWarrior lets the user generate an initialisation routine without knowledge of register definitions and peripheral interdependencies, which proved useful. Configuring the MCU without the GUI can be challenging as the datasheet can be unclear at times, i.e. the datasheet doesn't state clearly how to configure the internal clock to achieve a given instruction frequency, although this was easily done with the GUI, and the code it produced was commented. On the whole the Freescale MCU provided more control than both the Atmel and Microchip MCUs, i.e. more low power modes allowing different peripherals to be run while other parts of the MCU are shut off as well as offering much finer control of clock frequency.

Configuration of both the Atmel and Microchips MCUs was similar in that it involves following through the guidelines set-out in the respective datasheets. The Atmel datasheet provides code examples in both C and Assembler, while the Microchip datasheet provides code examples only in C. However, the Microchip datasheet was considered to be the easiest to read.

C. Programming

1) Microchip: Register definitions as defined in the HI-TECH compiler's include-files were the same as used in the datasheet. This simple expedient reduced confusion when writing code. The ability to address single bits in each register made it easier to deal with flags, control external hardware attached to a port, and configure peripherals such as the ADC without dealing with the whole register.

Real time clocking was required in this application but wasn't available on the Microchip MCU. While interrupts could be generated to emulate a real-time clock, they couldn't be generated in ideal fractions of seconds. One solution to this was to trim the reference clock frequency but this then affects other peripherals running from the clock.

Unexpected behaviour can occur when reading from, then writing to, adjacent bits on a GPIO port in quick succession. When you perform an instruction such as a bit-set on a register the PIC first reads the entire byte of the register, then it performs the operation on the number it has just read, and finally it writes the number back to the register. This is fine, except on ports. If you perform such a read-modify-write operation on a port register the MCU reads the actual state of the pins, rather than the output latch. If the port's output pins have yet to slew to their final levels from a previous, recently-executed write, the read gets the previous, not the "current" value. This period can be quite long if there is significant capacitive loading on the pin. In fact, with sub-microsecond instruction times, normal capacitive loading can result in unexpected operation, as shown in figure 3.

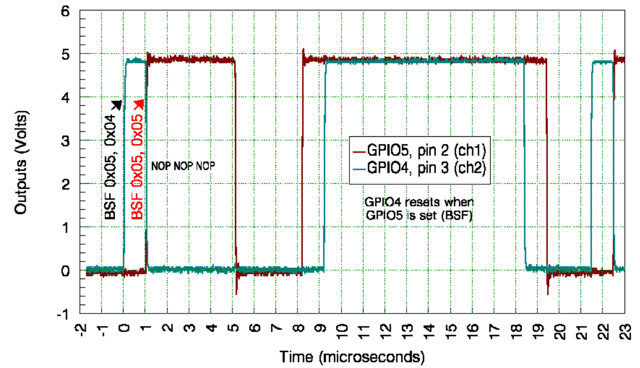


Fig. 3: Measured output signals on a PIC MCU with sequential bit operations acting on different bits applied to an output port register. Note that GPIO4 is reset to a previous value when the second instruction tries to modify only GPIO5.

Using the EEPROM was straightforward as routines were provided by the HI-TECH compiler to handle both the reading and writing of data.

With a set stack size of 16 words, the Microchip had limited ability to move through several sub-routines. Subsequently, recursion would quickly result in stack overflow, and is not supported by the compiler. This limitation was not present in either of the other MCUs.

2) Freescale: CodeWarrior provided soft definitions for accessing individual bits within registers, however these weren't as easy to follow as the Microchip MCU as the definitions were included in a separate file (as opposed to being directly in the datasheet).

The Freescale had real-time clocking, which was easily set-up by the GUI with a count value that could be modified if the clock was found to be to fast or slow.

Since the Freescale MCU doesn't have EEPROM, flash memory had to be used for all non-volatile storage. For those unfamiliar with the differences between flash and EEPROM, flash must be erased in blocks (called pages) but can be written byte-by-byte. The restriction is that when writing a byte to flash, the destination byte must have previously been erased which means erasing an entire 512 byte page of memory. Basically, the Freescale lets you use some of your leftover code space to store your own variables (provided your code doesn't need to use that space while operating). Configuring both the compiler and MCU to allow writing to flash from within the code was not adequately documented and required a lot of trial and error to get working. To achieve flash writes a function that copies the data from one location into flash must be loaded into RAM, which is done with another function. The compiler required directives to keep track of where the functions were to be written when the MCU was programmed and then where it would end up at execution time.

3) Atmel: Unlike the Freescale and Microchip, the Atmel did not offer the ability to access individual bits within a

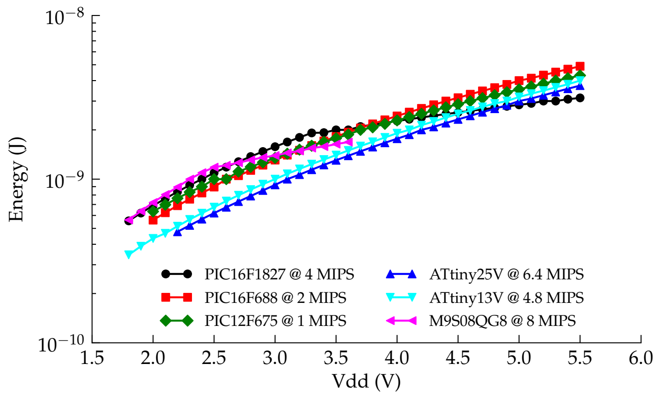


Fig. 4: Energy consumed per instruction cycle for six low power 8-bit MCUs

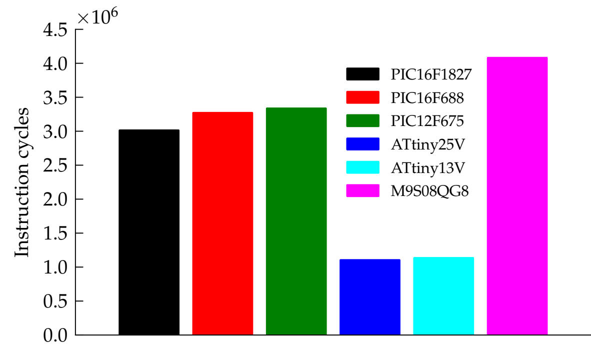


Fig. 5: Number of instruction cycles required to complete the exact same function

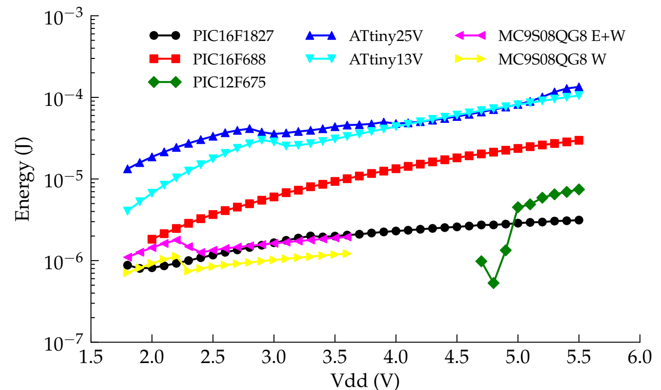


Fig. 6: Energy cost of writing to non-volatile memory.

register. This meant that it was up to the user to implement their own macros for setting and clearing bits, which can be daunting for those not used to defining their own macros. This isn't necessarily a negative: Not relying on inbuilt macros leads to greater code portability and a slight performance increase when setting or clearing multiple bits in succession.

While the Atmel did not have a real time clock, it had a comparator that could be configured to work with a timer. This meant the timer would count to a value set in the appropriate register before resetting and generating an interrupt. This count value could be modified if the clock was found to be too fast or too slow.

D. Efficiency

In a separate investigation, the energy efficiency of three MCUs from the same three manufacturers was investigated. The MCUs used in that investigation where lower pin count versions similar to the ones compared in this case study. Figure 4 shows the amount of energy each investigated MCU consumed per instruction while operating at their most power efficient (in terms of instructional efficiency) point. This graph shows so much overlap between not only models but manufacturers that one could conclude that these results are representative more of the limitations of microprocessor fabrication technology than processor design. This indicates there are no clear winners in terms of energy per instruction when each chip is operating in its highest efficiency state.

However, figure 5 shows that the number of instruction cycles each chip takes to complete the exact same function can vary significantly between MCUs. This graph is independent of clock frequency and has already taken into account that the Microchip MCUs only complete one instruction cycle for every four clock cycles. These results suggest that the Atmel ATtiny88-AU running at 12 MHz can execute code more than twice as fast as the Freescale MC9S08QB8 running at 20 MHz and over four times faster than the Microchip PIC16LF1938 running at 32 MHz.

The amount of energy required to erase and write to non-volatile memory was also measured and compared. The results of these measurements are shown in 6. This shows that the Atmel MCUs consumed, at times, over ten times more energy when writing to non-volatile memory than the Freescale and Microchip MCUs. In the case of the Freescale MC9S08QG8, which has flash instead of EEPROM, there are two traces shown. Since flash memory must be erased in blocks (pages), the trace labelled 'MC9S08QG8 E+W' has been calculated by taking the total energy cost of erasing a page (512 bytes), dividing that by 512 and adding it to the amount of energy required to write one byte. The trace labelled MC9S08QG8 W represents the amount of energy used to write one byte assuming the destination byte has already been erased

IV. Conclusion

Apart from the high cost of the development kit the Freescale MCUs offer fantastic value for money and is a clear winner for low cost applications not requiring EEPROM. The Freescale MCUs tend to offer greater control for the programmer, such as the different levels of sleep, highly configurable clock frequency, a choice of using pages of flash

for code space or program variable storage. This flexibility and different architecture mean the Freescale is more suited to professionals although the GUI configuration interface helps alleviate issues for new users.

For applications requiring non-volatile memory storage, or where program size is more important and execution speed less so, the Microchip range is a sensible choice. The documentation is clear, the IDE is simple and when used in combination with the PICKIT programmer offers a quick and easy way to compile and download code. The bundled version of the compiler is adequate for almost all situations and the Microchip tended to be more power efficient when writing to non-volatile memory and whilst in standby mode.

The Atmel MCUs offered very high processing efficiency and documentation comparable to that of the Microchip. It is less suited to the tap-timer project than the Microchip MCU as the extended supply voltage range and high code execution performance aren't necessary for this application. The documentation included useful code examples in both C and Assembler and there is a large online community based around Atmel MCU development.

The verdict is that the Microchip MCU was the most suited to this particular project due to its ease of use, clear documentation and non-volatile memory. For medium-high volume manufacturing the Freescale would be the first choice simply due to cost. However, this would require redesigning the software to accommodate the lack of EEPROM or use the flash effectively. For projects requiring more computational power the Atmel would become more favourable. Microchip offer a very large range of MCUs, each suited to different applications. This can mean it would be possible to select a Microchip MCU that is better suited to specific project requirements than the more one-size-fits-all approach taken by Atmel.

V. Acknowledgements

The authors are indebted to a number of contributors. Punit Solanki was observed tackling Atmel microcontrollers for the first time. Shabir Azizi provided some circuit designs and sprinkler control theory. Michael Cosgrove, Pawan Srestha and Weiqian Zhou provided hardware and software support. One of the authors was supported by a Waikato University scholarship.

References

[1] T. Honold, "Seventeen steps to safer c code," *Embedded Sysems Design*, vol. 24, no. 4, p. 16.

[2] M. Jackson, "Representing structure in a software system design," *Design Studies*, vol. 31, no. 6, pp. 545–566, 2010.

[3] J. Jacky, *The Way Of Z: Practical programming with formal methods*. Cambridge, New York: Cambridge University Press, 1997.

[4] D. Doron, "Creation and validation of embedded assertion statecharts," S. Man-Tak and D. Kadir Alpaslan, Eds., vol. 0, pp. 17–23.

[5] H. Warnars, "Object-oriented modelling with unified modelling language 2.0 for simple software application based on agile methodology," *Behaviour & Information Technology*, vol. 30, no. 3, pp. 293–307, 2011, iSI Document Delivery No.: 762IT Times Cited: 0 Cited Reference Count: 18 Warnars, H. L. H. S. Taylor & francis ltd Abingdon.

[6] G. Stefania, "Formal test-case generation for uml statecharts," L. Diego and M. Mieke, Eds., vol. 0, pp. 75–84.

[7] S. Tim, "Transformation of uml state machines for direct execution," M. Wolfgang, Ed., vol. 0, pp. 117–124.

[8] G. Pint, "Impact of statechart implementation techniques on the effectiveness of fault detection mechanisms," I. Majzik, Ed., vol. 0, pp. 136–143.

[9] J. K. Peckol, *Embedded systems : a contemporary design tool*. Hoboken, NJ: John Wiley & Sons, Inc, 2008.

[10] T. Wilmshurst, *An introduction to the design of small-scale embedded systems*. Basingstoke: Palgrave, 2001.

[11] "Software documentation - http://en.wikipedia.org/wiki/software_documentation," 29/06/2011.

VI. Appendix—UML Diagrams

Documentation ensures that the pre-determined specifications of the project are met, which is important as this is often cited as one of the primary reasons why software projects fail [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. To document this project UML (unified modelling language) was chosen. Figure 7 shows the overview page of the UML documentation. Figure 8 summarises the entire set of diagrams required for the "smart" sprinkler. While UML is not yet easily capable of generating code from a diagram, it offers an easy to comprehend, a visual representation of the work-flow of a piece of software, including the inter-dependencies of tasks. It should be noted that while the UML documentation for the project is very complete in outlining the functions, it does not give clues in how to implement them in software.

Figure 7 shows the outline of the code with main and interrupt routines in two swim lanes. Figure 8 presents the entire set of UML activity diagrams defining the smart sprinkler code.

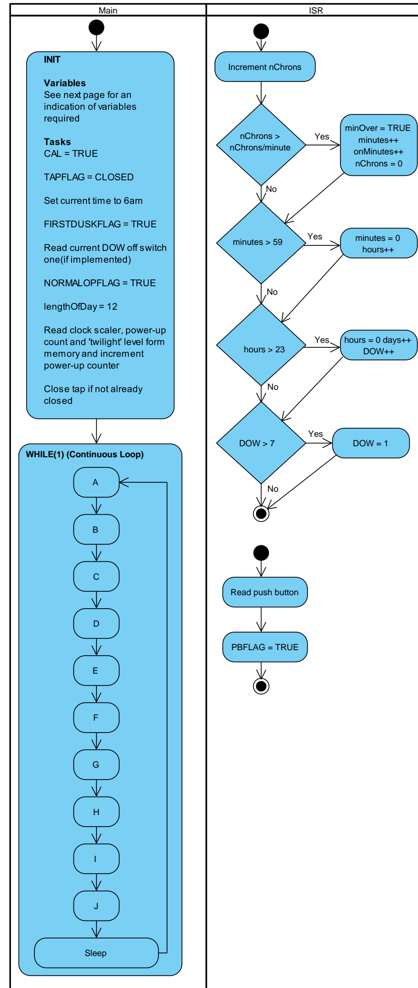
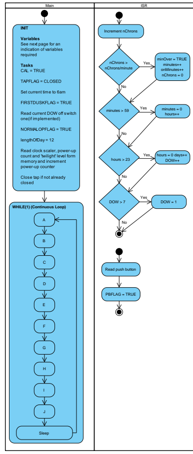
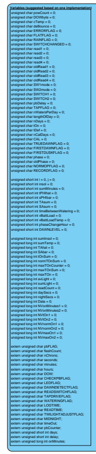


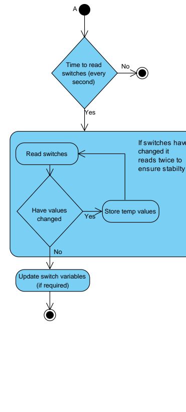
Fig. 7: An example of a UML activity diagram for the "smart" sprinkler project, the right-hand swim lane illustrating the Main line, the left-hand swim lane illustrating the interrupt service routines, handling time keeping and push button events.



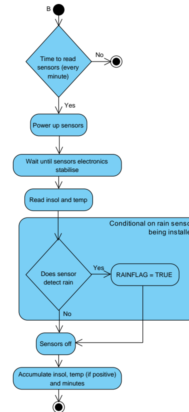
(a) Main & ISR Routines



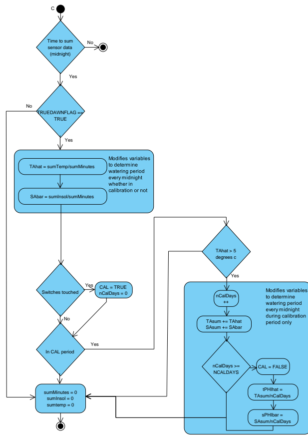
(b) Suggested Variable Names



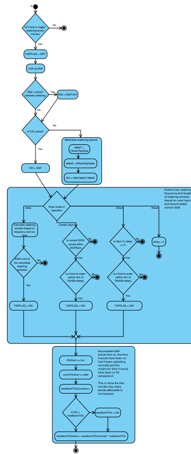
(c) Switch Reading Routine



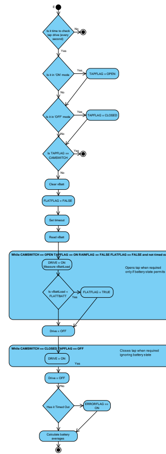
(d) Data Collection Routine



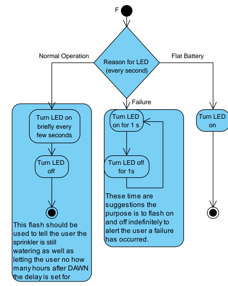
(e) Data Interpretation Routine



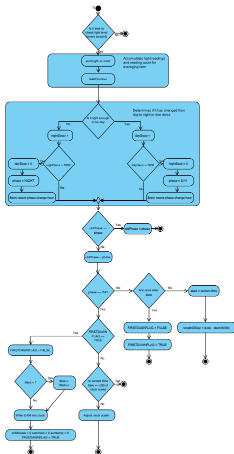
(f) Water Triggering Routine



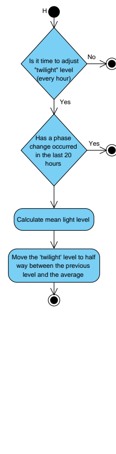
(g) Tap Drive Routine



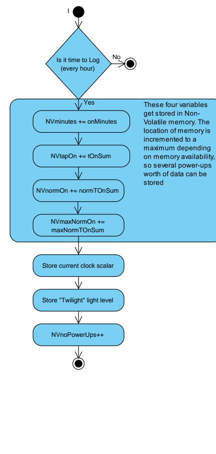
(h) LED Flash Routine



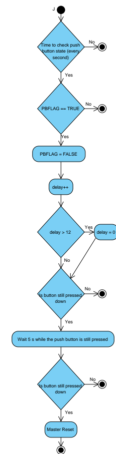
(i) Solar Synchronisation Routine



(j) Twilight Adjustment Routine



(k) Non-volatile Data Logging Routine



(l) Push Button Handler Routine

Fig. 8: The complete set of UML diagrams documenting the "smart" sprinkler firmware. When reading an electronic copy of this manuscript full detail can be seen by zooming in as required.

Smart Sprinkler Operating Manual

The Smart Sprinkler Tap is a self-contained garden sprinkler that requires no complicated installation and features a simple user interface. It will typically run for a year on two AA batteries. It automatically spreads waterings throughout the day, always starting at dawn. Once set up to water your plants you can forget about wasting water or plants getting insufficient water if the weather gets hotter. The Smart Sprinkler Tap automatically adjusts the duration of watering to suit the growing conditions!

LED Indicator

In normal operation the LED will flash a "heartbeat" like a smoke alarm to assure you that it is working. If a delay has been set (see below), the LED flash count corresponds to the time delay set by the user. If the battery begins to go flat, the LED will remain on. Rapid flashing indicates a hardware failure.

Installation

Set up your irrigation pipe and sprinkler heads as desired. Connect the Smart Sprinkler Timer to the tap, and the irrigation pipe to the outlet of the Smart Sprinkler Timer. Turn on the water. Install the batteries. The Smart Sprinkler Timer will open and immediately close to confirm correct operation of the valve mechanism.

Setup

Programming Frequency

To set the frequency of watering adjust the first dial accordingly. Choose either watering several times a day (12,8,6,5,3,2,1), or watering every several days (1,2,3,7).

In some markets the Frequency dial will be labelled with certain days on which it is permitted to water. In this case, before installing the batteries adjust the day dial to the current day of the week so the Smart Sprinkler knows what day of the week it starts on. Then adjust the dial to set the appropriate days to water.

Programming Duration

To set the duration adjust the second dial accordingly — choosing the duration in minutes. ("OFF", 1, 3, 5, 10, 15, 20, 30, 60, 90, 120, "ON"). In the first few days of operation, you should adjust the duration to suit conditions, so that plants get enough water, but not too much. Once this adjustment process is complete, do not touch the controls. The Smart Sprinkler will make those adjustments for you. If you change the settings, the timer will assume you have made changes to your irrigation system, and will water as you ask until a few more days go by. When left alone, it starts to make its own changes.

If you wish to prevent all watering select the "OFF" position.
If you need to turn on the water supply to maintain or adjust the irrigation system, selct the "ON" position.

Adjusting Delay After Dawn

If you have several Smart Sprinklers in the one garden, you may want to delay the start of watering on some of them so that they do not all go off at the same time. A one-hour delay may be requested by pressing the recessed push button. To adjust the delay after dawn at which the sprinkler is to begin watering press the push button once per hour of delay[1]. The heartbeat flash counts out the delay of each individual timer. If for any reason the sprinkler needs to be reset simply hold the push button down for 5 or more seconds or alternately remove and replace the batteries.

[1] The maximum delay after dawn is 12 hours, if the push button is pressed more than 12 times the delay will be reset to zero.