A Visualiser for Embedded Systems

Development of a visualiser to support novice learners' understandings of embedded systems

Bill Collis

A thesis submitted in fulfillment of the requirements for the degree of Master of Engineering, The University of Auckland, 2014.

Abstract

Over a period spanning more than ten years I developed a programme of learning in the area of embedded systems for secondary school students at Mount Roskill Grammar School. This programme currently involves 12 classes totalling 350 students. Learning is centred on the ATMEL AVR range of microcontrollers and forms part of students' studies in Technology Education. It is taught by four teachers some of whom do not have strengths in electronics and programming. Resources I have developed to support students and colleagues include a website, a textbook and a computer program called 'System Designer'.

For a number of years there has been a developing awareness of issues concerning my own students' understandings of software for embedded systems which are confounded by the hidden nature of embedded systems and the disparate nature of embedded systems development. This research presents a journey towards a deeper appreciation of students learning needs and the development and testing of a novel visualisation tool for embedded systems targeted at novice embedded systems learners.

Literature about the difficulties of novice programmers along with the learning needs of novices in the field of embedded systems was reviewed to inform this research. The context of this study is secondary school learning so other pertinent literature that informs learning in the classroom was also reviewed; this includes educational theory, pedagogy, model-based learning, research from engineering education, the New Zealand Curriculum and the changing metaphor for learning in New Zealand technology education from an acquisition to a participatory model.

To facilitate development of the visualiser a range of program visualisers and embedded systems simulators were analysed using research into best practices for model-based learning tools. Lexers and parsers were then written to create interpreters for both C and BASIC languages and the visualiser incorporating these was integrated into the existing System Designer application. Further the visualiser was developed as a staged learning tool to negate cognitive load issues for learners. To support colleagues who will use the tool a range of tasks have been developed and options to extend some features of the visualiser have been implemented as well.

The visualiser and some tasks were tested in a qualitative study with second year engineering students who were unfamiliar with embedded systems. Lecturers were also interviewed to gain their insights into the new tool. The goals of the study were to understand how novice

embedded systems learner's related to the visualiser as a model-based learning environment and secondly to identify its benefits for novice learners of embedded systems.

The study was positive for the visualiser as a model-based learning tool, and valuable for revealing errors and modifications before implementation at school. A visualiser is an analogy of a real system, as such it has limitations which experts recognise but which novice learners do not. A planned result was to explore limitations of the visualiser. One key limitation is well known and relates to when learners single step through program code rather than run it at normal speed; in this case understandings of the real-time and reactive nature of embedded systems are not fully appreciated by novice learners. While modifications to the visualiser were made to counter some of this effect, teachers must work within this limitation and not expect students to comprehend it for themselves.

For novice learners of embedded systems the visualiser showed significant benefits in terms of the interrelatedness of hardware and software, the concept of state and the reactive nature of embedded systems. Some of these understandings are inherent in the visualiser and some of them require careful choice of contextualised tasks for students to engage their understandings. Exploration of real-time and concurrent aspects of programming embedded systems is still required.

Programming thinking was an important part of this research because for students learning about embedded systems at school this is their introduction to programming as well. Literature into programming thinking revealed a range of surface and deep understandings that novice programmers need to develop as well as the importance of engaging students in their own learning via the tasks given to them. The results of testing showed how the visualiser and the tasks used with students contribute to these understandings; specifically that deeper understandings require tasks with real-life contexts.

One significant outcome of this research has been how the literature review and the opportunity to engage in depth with a few learners has informed understandings about what students experience while learning; this has deepened my pedagogical content knowledge - the blend of subject matter with educational practice.

Key words: program visualisation, mental model, novice learner, embedded systems learning, learning to program

Acknowledgements

I would like to express my special thanks to:

The 1,000 students at Mount Roskill Grammar School over the last 13 years that I have had the privilege to work with on a daily basis and have inspired me to keep thinking about their learning.

The Ministry of Education, who have generously provided one year away from the classroom for me to read, reflect, investigate and explore the ideas captured in this research.

My supervisors, Dr Gerard Rowe and Dr Partha Roop for their wisdom and insight.

My wife Julie, my soul mate.

A number of software libraries and resources were used in the development of System Designer and the visualiser, and contributed to making the application possible:

- Mindfusion Windows Forms software library
- Avalon Dock software library
- Fast Colored Text Box software library
- The articles by Jack Crenshaw (1988) were invaluable when writing the parsers

Table of Contents

List of Figures

List of Tables

Chapter 1. Te wero – (the challenge)

"You'll never get kids to learn that stuff", one of my colleagues blurted out. I had just ended a presentation on a course I was developing for microcontroller programming and interfacing as part of my secondary school technology teaching course in 2000. Many teachers like my colleague believe that some things are too difficult for high school students to learn. In my classroom practice I have found the opposite to be true; and as Barak (2002) found, the demanding nature of learning in electronics actually promotes student interest whereas lack of challenge stifles it.

Significant challenges nonetheless do exist for learning in embedded systems. They present significant pedagogical challenges for teachers such as how to cope with the multitude of new skills that students require, the non-trivial theory needing to be learnt and the significant range of new concepts for students to develop (Doboli, Doboli, & Currie, 2008). In addition the learning processes best undertaken by students in embedded systems are different to the way schooling has often been experienced by students and teachers; as success in this area is correlated with understanding which does not come from rote learning and low-level work (Blumenfeld et al., 1991). In schools there are resource challenges also; these are not so much related to hardware needs but that the growing interest in modern technology places pressure on the current cohort of technology teachers whose backgrounds are not in this area of technology.

As the curriculum leader for this area at school I have responded to these challenges by developing a range of resources to support both colleagues and students. These include: a resource website, a textbook and a software application called 'System Designer'. These resources have been developed, trialled and tested in an ongoing manner with students since 2002. Each time I think I have come a step closer to understanding students' learning needs I have uncovered yet another issue that demands my attention. Currently I have come to a point where I need to address some deeper facets of students' understandings with regard to embedded systems and I have the privilege of having a one year study scholarship during which I have the opportunity to explore in detail research about the needs of novice learners when beginning to learn about embedded systems.

This thesis begins with a wide ranging exploration of the needs of novice embedded systems learners. Learning is a complex area and while only the most salient aspects are covered this is still wide ranging. In Chapter two this covers: identity, what embedded systems are; legitimacy, why they should be taught; and selection, what specifically should be taught. This chapter includes a discussion on the changes that have taken place in Technology Education in New Zealand as these changes have had significant bearing on my development and classroom practice. Chapter three draws on the information in Chapter two to cover how embedded systems can best be learnt and describes the powerful benefits and limitations of learning using both real hardware and visualisation as learning tools. In Chapter four visualisations that I have developed for use with students are discussed to support the direction for this work. Chapter five discusses the choice of features to be developed into the visualiser by analysing existing visualisation tools within a framework of understandings from literature using best characteristics of model based learning tools. Chapter six discusses the development of the supporting pedagogical materials developed for use in the classroom. Chapter seven discusses research methodology and chapter eight the results of a pilot study with students conducted by me including interviews with pertinent lecturers. Chapter nine is the results discussion and directions for future work.

Research questions asked were:

- RQ1. How successful is the visualiser as a model based learning environment?
- RQ2. In what ways does the visualiser benefit novice learners of embedded systems?

Appendices cover the technical details of development of the visualiser and the features of its parent application, ‘System Designer’, as well as a brief introduction to the electronics programme of learning at Mount Roskill Grammar School.

Chapter 2. Embedded systems and learning

Learning is a multifaceted area of study as it is about learners' knowledge, skills and understandings as well as the teacher and their understandings of learning and curriculum. This chapter seeks to explore the important aspects of embedded systems education, and relate these to secondary school education. This includes: a range of characteristics of embedded systems, trends in technology education, the New Zealand Curriculum for schools, novice programming issues, learning theory, pedagogical practice, and engineering education. To integrate these broad topics together the didactic model of embedded systems learning by Grimheden & Törngren (2005) is used. This has four themes: identity, what embedded systems are; legitimacy, why they should be taught; selection, what specifically should be taught; and communication, how they should be taught. The first three are covered in this chapter and the fourth in Chapter 3.

2.1. Identity - what are embedded systems?

Embedded systems is a vast field; Koopman (2005) identified 12 different areas for study based upon the applications of embedded systems. One of these, small and single microcontroller applications, dominates the secondary school learning environment, although robotics, control systems, networking, and wireless data will also be encountered in schools. While an embedded systems graduate engineer would develop deep understandings of one or more areas, this research explored introductory understandings for novice learners at secondary and tertiary levels.

2.1.1. They have purpose

The first of these is to revisit the learner's frame of thinking about what a computer is. At a surface level a new learner of embedded systems may compare them to the common computer (desktop computer, laptop, notebook or tablet) they are accustomed to, by their small size, trivial technical specifications and simplistic human interfaces. While the common computer is often compared to another common computer via its memory size or processor speed the same comparison with an embedded system would make it appear trivial, as an embedded system may operate ten thousand times slower and have access to ten million times less memory than a common computer. The central understanding is that an embedded system is built to serve just one purpose so cannot be compared to another by specifications because its legitimacy, or justifications for cost and technical specifications, comes only from fitness for purpose within its the unique environment (Grimheden & Törngren, 2005; Vahid & Givargis, 2002). The concept

of context and fitness for purpose is a significant understanding required for embedded systems developers.

2.1.2. They are ubiquitous

While the computer appears in almost every aspect of our lives, embedded systems are however more ubiquitous, they are the small computers found inside toys, consumer appliances, cars, industrial controllers, aircraft and the robots that make all these devices. In 2012 2.2 billion desktop computers, laptops, tablets and mobile phones were produced (Gartner Research, 2013). While no actual figures for the number of embedded systems during the same period of time is known an estimation of 15 billion can be made based upon the number of microcontrollers manufactured. Gartner Research rank Microchip as having 6.7% of the market (The Databeans Monthly, 2012) and during 2011 Microchip announced that they had shipped one billon microcontrollers in the previous 12 months (Giovani, 2011), which approximates the number of microcontrollers produced in 2011 up towards 15 billion, almost seven times the number of common computer devices. Secondly to this, while the common computer has a short life (one to three years) an embedded system may continue to function for many years so the numbers of embedded systems may be growing at a rate much faster than the common computer system.

2.1.3. Embedded systems operate differently

An early understanding is to stop seeing computers as transactional in nature (Koopman et al., 2005). The common computer system is transactional as the user powers it up and chooses which programs to run and when; then when the user is finished the computer is powered down. In contrast the embedded system is an automaton; it works automatically from power up following a predetermined single program operating without any user interaction at all and in many cases will never be powered down.

Embedded systems are reactive and operate within real-time. Where the common computer is judged against its performance relative to a user's needs, the embedded system is in a constant relationship with its environment (Benveniste & Berry, 1991) and judged in terms of its ability to react and respond in time with the dynamic needs of the environment (Tuma & Fajfar, 2006; Winzker & Schwandt, 2011). This leads to the need for an embedded system to operate concurrently or to provide the illusion of doing many things at once; in a single processor system this requires the ability to rapidly context switch.

The dynamic nature of environment leads also to the concept of 'state' in an embedded system. While in a common computer system one event is unrelated to either the previous event or the events that follow, in an embedded system the present condition of the system is a consequence of past conditions and will have direct influence over the future condition of the system (Lee & Seshia, 2011).

Embedded systems may often distribute or share system processing. While the common computer is usually networked with other computers, it most often operates as a stand-alone device or in a subservient role to a centralised server. In a distributed system several or many locations undertake control actions and interact with other systems such as in public transport control systems, industrial plants or across a whole aircraft. A result of this is the increased understandings required due to many varied interface and bus specifications in use, along with varied data protocols and having an overview of how the many subsystems interact so as to avoid issues with latency and conflict of control responsibilities.

2.1.4. The interrelatedness of hardware and software

While it is the programs in both common computer and embedded systems that allow their power to be realised, there are new realisations for those who write embedded systems software. In the common computer there are layers of software which abstract the hardware away from the sight of software developers; this is not the situation however when writing software for an embedded system. So while in a common computer system the hardware and software can be seen as independent, in an embedded system these layers do not always exist so hardware and software cannot be dealt with independently (Bertels, D'Haene, Degryse, & Stroobandt, 2009). Software development in an embedded system means becoming familiar with how to use the complex array of hardware timers, interrupts and bus interfaces found inside modern microcontrollers. Consequently an embedded systems developer must know a great deal about both hardware and software.

At a deeper level an embedded systems developer must have more than knowledge about both hardware and software; they cannot function effectively without an understanding of the interrelatedness of the two (Kossiakoff, Sweet, Seymour, & Biemer, 2011; Vahid & Givargis, 2002; Wolf, 1994). Developers must know how software design affects hardware decisions and how hardware constraints affect software design. A lack of a understanding of the unified nature of hardware and software leads to design errors that are often not caught until the implementation stage and are often extremely costly to fix (Janka, 2002).

2.1.5. They require additional tools

Further adding to the complex life of a developer of embedded systems is the need to learn about development and debugging tools which use 'third-party' computer equipment separate to the embedded system itself (Winzker & Schwandt, 2011). This is required because many interact solely with the environment and have no human input or output devices such as keyboards and displays.

To support embedded systems in reliably meeting the real-time constraints of their environment, real-time operating systems (RTOS) have been developed to help developers keep on top of concurrent processing requirements. These however require their own understandings regarding interrupt processing and issues such as race conditions, dead locks and critical sections of code.

2.1.6. Validation of operation

Contextualised understandings of embedded systems are summed up by concerns about how we have given up responsibility for the well-being of people and environments to embedded systems controllers. In some cases this may have minimal consequences such as if the controller for an electric window in a car was to fail. However if the fuel control unit in an ambulance or aircraft engine were to fail it might have mission or life threatening consequence. With our increasing reliance on embedded systems the understanding of how a system can be reliably checked and verified under all its varied inputs is an imperative understanding for developers (Kern & Greenstreet, 1999).

A deeper understanding of verification is that simulation and testing is unsatisfactory in large and complex systems that serve dynamic environments as it is impossible to check every change with every requirement; this leads to formal mathematical verification methods of the temporal (time-related) properties of liveness, reachability and safety (Sinha, Roop, & Basu, 2014).

2.1.7. They are not without significance

One further issue about embedded systems is significant for our society: the issue that embedded systems are seldom seen and remain hidden from our view. They are a 'secret' technology, unrecognised by the majority of the population as they silently regulate the fuel in our car engines, determine how long our clothes are washed for and manage on our behalf a myriad of other tasks that we have given over responsibility for. The deep understanding required in

learning about embedded systems is this hidden and unrecognised role of control that embedded systems have over our lives.

2.2. Why should embedded systems be taught in schools?

The description in the last section of what constitutes an embedded system initially seems to shout (as my colleague once did) "this is too hard to teach in schools!" The New Zealand Curriculum and the aims of technology education nevertheless provide highly valid reasons for learning about embedded systems. This section discusses the metaphors of learning in depth, their relation to the curriculum and the place of learning about embedded systems in New Zealand technology education.

2.2.1. The New Zealand Curriculum

The New Zealand Curriculum (Ministry of Education, 2007):

> *"Our vision is for young people: who will be creative, energetic, and enterprising; who will seize the opportunities offered by new knowledge and technologies to secure a sustainable social, cultural, economic, and environmental future for our country; who will work to create an Aotearoa New Zealand in which Māori and Pākehā recognise each other as full Treaty partners, and in which all cultures are valued for the contributions they bring; who, in their school years, will continue to develop the values, knowledge, and competencies that will enable them to live full and satisfying lives; who will be confident, connected, actively involved, and lifelong learners."*

While there is a surface link between "new knowledge and technologies" and the learning of embedded systems, the curriculum contains no prescription for student learning (Compton, 2007) and devolves responsibility for detail onto schools and communities (Ministry of Education, 2013). This demands more than a cursory view of the curriculum; it requires schools and teachers to have a detailed appreciation of the fit of the subject with the curriculum's focus on the real world. The curriculum statement gives broad guidance by stipulating three underpinning requirements of learning; namely: knowledge, competencies and values.

2.2.2. Knowledge and competencies

Knowledge and competency is expressed in the predominant metaphor of 'acquisition' (Sfard, 1998); where students are seen as 'acquirers' of both knowledge and competencies in the process of their learning. A metaphor represents the approach educators take on learning based upon their epistemology or understanding of knowledge. Three significant educational theories underpin the acquisition metaphor; these are explained as they have relevance to the New

Zealand Curriculum and the changes that have taken place in technology education and consequently embedded systems education in secondary schools.

Behaviourism or response acquisition is a theory of learning where students are seen as outputs of a system and passive receivers of knowledge (Manus, 1996; Mayer, 1992). What students need to know is provided for them, along with stimuli and feedback (consequences or reinforcement) to elicit correct responses (Ertmer & Newby, 2013). Behaviourist approaches however lead to tightly constrained understandings that students cannot apply to novel situations as shown in Figure 2.1.

While change in technology education began in 1975 and the behaviourist approach was completely removed in the 1995 curriculum, many technology teachers are still stuck in their behaviourist traditions (Harwood & Compton, 2007; Harwood, 2002; Mawson, 1998).

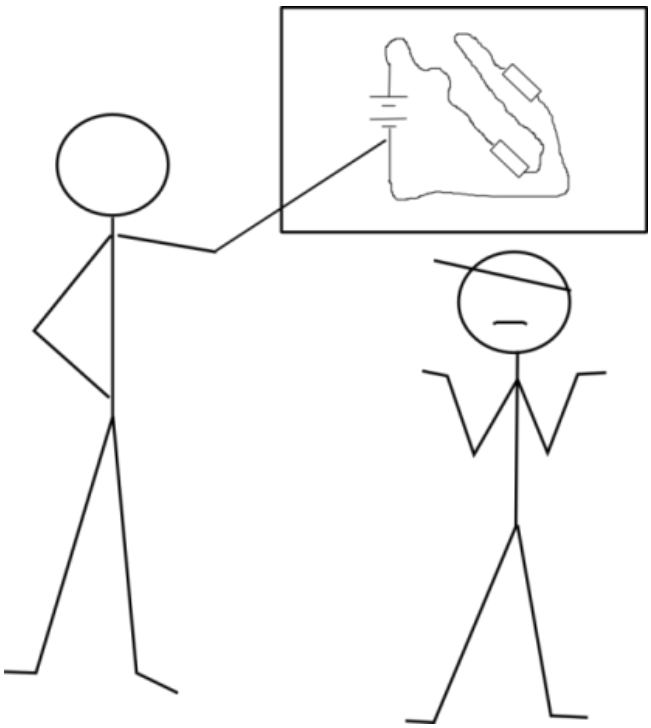


Figure 2.1: Behaviourist practices lead to poor understanding in unfamiliar situations

A further stage of thinking about learning is that of cognitive acquisition by the learner (Case, 2008; Mayer, 1992). Learning cognitively is developing knowledge structures in the mind, often called schemas, that once learned can be transferred into novel situations. The learner is an information processor and the teacher's role is "to increase the amount of knowledge in the learner's repertoire" (Mayer, 1992, p. 407) as in Figure 2.2. The learner is however not a passive

receptor of learning but an active agent in the process of learning, organising and building understanding (Ertmer & Newby, 2013). As opposed to behaviourism, cognitivism uses measures of understanding as the feedback loop from student to the instructor; in this way the learner has some limited influence on the teacher and their methods.

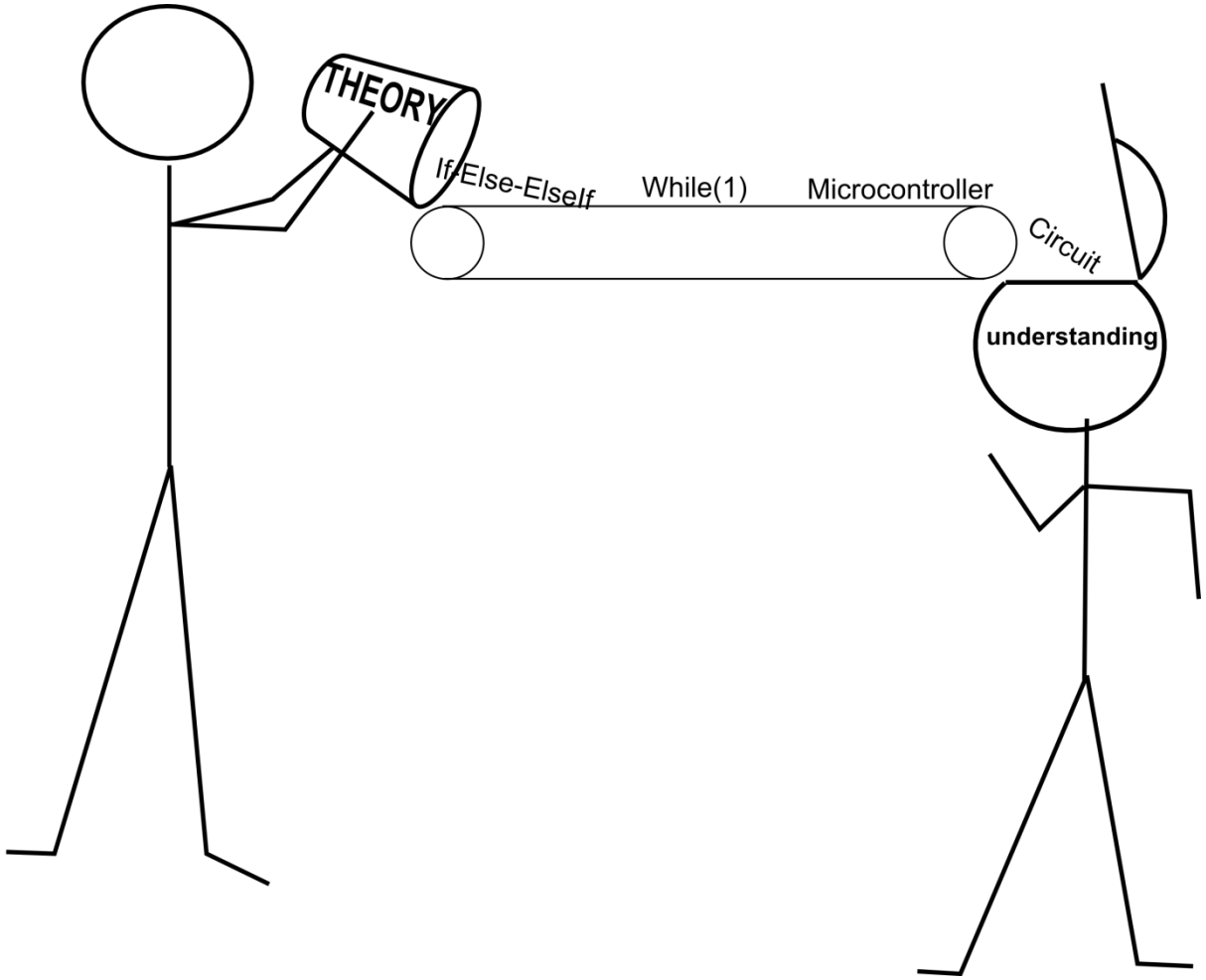


Figure 2.2: A cognitive or information processing view of understanding

Constructivism is a third learning theory of the acquisition metaphor. In constructivist education students are not just active participants but ultimately the only active agent in the learning process. Knowledge and understanding come from personal experiences, and are ultimately a student's own "private property" (Sfard, 1998, p. 6). Teachers become facilitators, relinquishing control of the learning process (Ben-Ari, 1998; Magrini, 2009; Perkins, 1999) as in Figure 2.3.

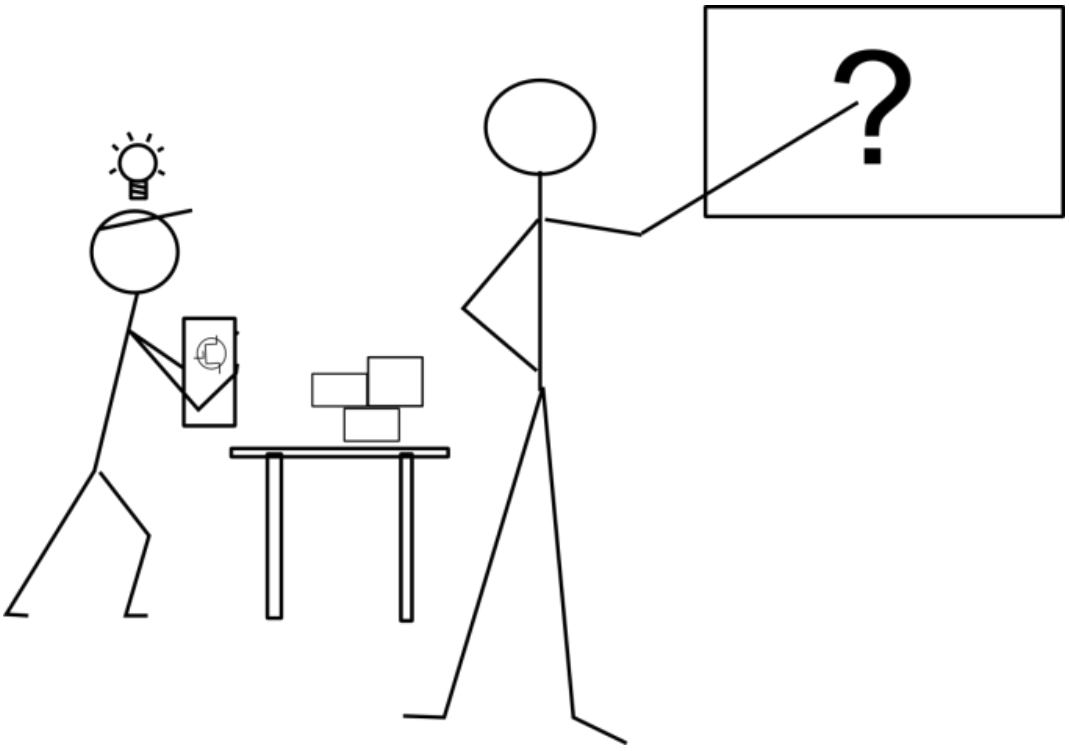


Figure 2.3: Constructivist learning through problem-based tasks

At a tertiary level the change to constructivism is seen as critically strategic for increasing student understanding, that it will reduce the downward trend in engineering student populations and provide graduates with the skills demanded by industry (Felder, 2012; Prince & Felder, 2006). Whilst seen as a paradigm shift by some (Ben-Ari, 1998), it is argued that a paradigm shift requires a "deeper level of correction" (Liu & Matthews, 2005, p. 389). Constructivism does not represent a new shift in paradigm as it does not move away from the acquisition metaphor which is focused on the individual. In technology education the significant aspect of constructivism that precludes it from being a paradigm shift is that learning outcomes are determined prior to the learning taking place (Harwood & Compton, 2007).

2.2.3. Values education and the metaphor of participation

The New Zealand Curriculum goes beyond knowledge and competencies as it articulates a range of values; so it is ontological as well, it is about the way people are built (Packer & Goicoechea, 2000). Whilst some individual values are mentioned in the curriculum such as excellence and curiosity, the majority are social values: community, participation, responsibility, sustainability, accountability, diversity, respect and justice (Ministry of Education, 2007, p. 10). This reflects the focus away from knowledge that someone owns (Johri & Olds, 2011) towards an education of acculturation into the norms of a community, fitting the African proverb that "it takes a village to raise a child". In educational theory this is the metaphor of 'participation' (Case, 2008; Johri & Olds, 2011; Sfard, 1998); and described by Vygotsky's socio-cultural learning theory (Liu & Matthews, 2005; Vygotsky, 1987) and by situated learning theory (Lave & Wenger, 1991). In this metaphor of learning full meaning and understanding only come from social participation in community as for the learner depicted amongst his wiser companions in Figure 2.4. For educational institutions this means that learning outcomes must be linked to the real world through actual life situations; the curriculum reflects this not only by being non-prescriptive but by substantial use of the word 'context' stipulating there needs to be: range, variety, social, cultural, complex, meaningful, diverse and relevant contexts.

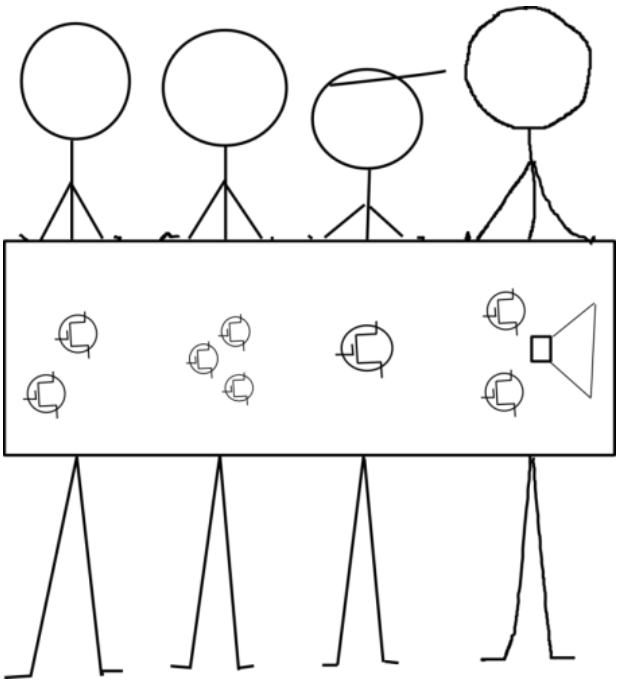


Figure 2.4: Participatory (socio-cultural) paradigm of learning

2.2.4. Technology education in New Zealand

In New Zealand the philosophy underpinning technology education is the participation metaphor and is captured by the values term 'technological literacy' (Compton, 2004). This is a term which lacks definition in the curriculum (Compton & France, 2007), as it is still evolving (Compton, 2007). It is problematic to define because it has at its root 'literacy' which is about more than knowledge and capability (Ropohl, 1997); it is about empowerment (Freire, 2004), volition (Custer, 1995) and freedom from control (Marshall, 1988).

Without a clear definition an example expresses the concerns that technological literacy addresses. There has been a change of computer education in schools over a period of time where computer science education has been replaced by a "computer driver license" (Freiermuth, Hromkovič, & Steffen, 2008, p. 216) and in doing so students have become passive and powerless consumers of computer technology, many now showing significant fear about its inner workings (Tuma & Fajfar, 2006). Learning in technology education is about combating this powerlessness by providing students with a mind-set of confidence that will allow them to actively participate in the technologies we are creating in our world.

There are three strands to technology education: Technological Knowledge, Technological Practice and Technology and Society and each has associated sub-components (Figure 2.5). These are explained in the subsequent sections and how embedded systems learning aligns with each.

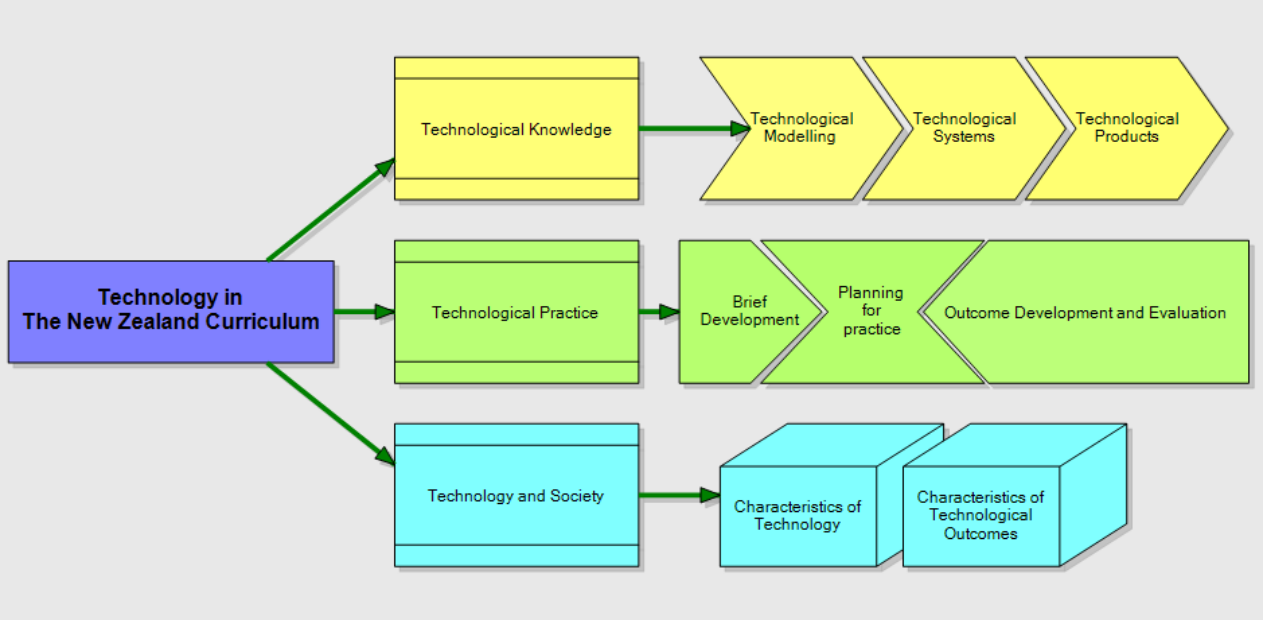


Figure 2.5: The three strands and eight components of Technology in the NZ Curriculum

2.2.5. Technological knowledge.

Of the three strands of technology that of knowledge seems the most easily recognisable; around us are endless technological devices that embody the knowledge used to create them. The nature however of this seeming endlessness makes it difficult to separate what is technological knowledge and what knowledge from other disciplines is used in technology. Compton and France (2006), the authors of the knowledge strand, analysed modern practices of technologists to abstract the components of knowledge distinctive to technology. The three components formulated were knowledge of: systems, modelling and products. Systems knowledge encompasses understandings such as subsystems, control, redundancy, systems parameters and the concept of a 'black box' that has inputs and outputs and a clear function, but hidden detail. Modelling incorporates the aspects of exploring function as part of design, analysing risk via testing prior to and during development. Products knowledge encompasses the roles, properties and performance of materials within technological products.

2.2.5.1. Embedded systems and technological knowledge

The congruence of learning in embedded systems with knowledge of the systems and modelling strands is natural; students model and test circuits using breadboards, they model embedded software using state machines and while developing a prototype continually test software. Students must understand the multiple facets of embedded devices from a systems point of view, often creating software adapters for black boxes they use in their work such as ultrasonic sensors and liquid crystal displays so as to work with them. In learning involving embedded systems school students would not often manipulate the composition or structure of materials as required by the products strand, although if a student were to create their own instructions for a soft processor in an FPGA this would suffice.

2.2.6. Technological practice

Complimentary to what technologists know is the way they go about their work, this is called technological practice, and is the tenet of the participation metaphor. This is no routine application of scientific knowledge (Custer, 1995) or a regurgitation of behaviourist training as it is not content knowledge but is process knowledge (Bohn, 1994). It is built from ongoing confrontation with open-ended problems and extends the understandings of constructivist learning where students "are basically left to find their own way" (Felder & Brent, 2004a, p. 276) by positioning the student inside a community of support. The 'company' software engineering

approach for students represents such a socio-cultural model with its high focus on process (Broman, Sandahl, & Abu Baker, 2012; Broman, 2010) as does the community of knowledge approach to a collaborative online textbook (Broman & Sandahl, 2011) and the development of open source software (Ben-Ari, 2004).

2.2.6.1. Embedded systems and technological practice

Technological practice has three components; the first is brief development which while front loaded is also an ongoing process of teasing out the key aspects of the environment and stakeholders needs. This parallels requirements engineering, and is at the heart of the highly contextual nature of embedded systems. This understanding of context is at the very core of embedded systems where half of the malfunctions come from misunderstandings of clients and environmental requirements (Broy & Stauner, 1999). In software engineering courses learning to elicit requirements has had significant impact on student success (Mohan & Chenoweth, 2011).

The second component of practice is planning, and is an incremental process of developing project management skill within students; the multi-faceted nature of even a simple school-based embedded system includes a veritable maze of hardware, software and environmental aspects that need managing, such as sensors, actuators and user interfaces as well as software design, writing, testing and debugging. Planning of resources including the most valuable asset of time is a core skill needed in embedded systems development.

The third aspect of technological practice is outcome development and evaluation. This is the practical application of methods and skills to realise something. Evaluation is the crucial culmination of practice where the outcome is measured against the specifications elicited by the technologist from the stakeholders and the environment and allows a judgement of the outcomes' fitness for purpose. Both are standard in embedded systems development, especially so, as evaluation in embedded systems development is so closely intertwined with clients, stakeholders, environments and purpose.

2.2.7. Technology and society

This strand of the curriculum is about recognising that technology and society are inextricably linked; that there is an increasing dependence and therefore vulnerability placed upon us from modern technologies, including those such as embedded systems (Robal, Kann, & Kalja, 2011). A technology in itself may neither be good or bad, but nonetheless it is "purposeful intervention" (Compton & France, 2006, p. 3) by humans so it cannot be completely value free. Technological

artefacts thus embody the power and control of the values of the designers (Warschauer, 2004) and their employers, the corporations that regulate design (Feenberg, 2010), through the limits placed upon us by their decisions. In technology education when the judgment of a piece of technology's fitness for purpose is extended to include all its social implications it is called fitness for purpose in the broadest possible sense.

2.2.7.1. Embedded systems and society

Embedded systems implicitly embody control over our society as they, for instance, automatically reduce the volume level when we plug headphones into a tablet or phone, operate the lid lock during the spin cycle of a washing machine, determine which DVD zones you can play and when a printer cartridge will automatically wear out; all based upon the preference of designers and not users. It is not inconsequential to society today to have so much unrecognised control everywhere around us. Technology education is about empowering users to appreciate the 'e-consequences' on our society of the choices made for us.

In technology education in the past there has been a strong focus on the take-home value of well-crafted products. As a technology educator I have a strategic reason for take home value in learning about embedded systems. This value is encompassed in the important discussions that students have with stakeholders about their embedded systems technology and a developing awareness of such technologies by the community that the student is a part of.

2.2.8. Summary of why embedded systems should be taught.

Much research has gone into establishing a national curriculum that guides education to help students become the best possible contributors to our society; and much research has also gone into abstracting what technologists do and building a curriculum for technology students that reflects technological processes. The goal with this section has not been to present embedded systems learning simply as congruent with the curriculum, but to realise that the nature of learning in schools must fit with what is taking place in the world; and the goals of education must align with the needs of our technological world. Modern society has technology at its core, especially the ubiquitous and hidden technologies of embedded systems, so secondary school education is made more relevant by addressing this technology.

2.3. Embedded systems - what should be taught?

Having established what embedded systems are and how the learning goals of the New Zealand Curriculum and Technology Education align with the technology of embedded systems, the next stage in developing a programme of learning was determining specifics about what should be taught; this took both a short and a long time. The short path was the surface knowledge of the course such as the choices for programming language, development tools and development platforms. This research pertains to the longer path: the on-going pursuit of student understanding, i.e. performance and capability. The bi-modal nature of these two paths aligns with my own growth from beginning teacher to a more experienced one, and typifies the learning journey from "ignorant certainty to intelligent confusion" (Felder & Brent, 2004a, p. 269).

2.3.1. The short path forward: methods, tools and programming language

In choosing a pathway for an embedded systems course at secondary school, the multidisciplinary nature of embedded systems could be approached by selecting a primary emphasis of one aspect such as hardware, software or system design such as many university courses often do (Greco & Nestor, 2011), or by choosing a combined overview course as some other universities do (Bertels et al., 2009). An overview course reflects the principle that the different disciplines in embedded systems are interdependent (Carstensen & Bernhard, 2007; Vahid & Givargis, 2002) and although they are many and varied, it is still possible to build a complete understanding if the equipment used is small in scale and unsophisticated (Bertels et al., 2009; Koopman et al., 2005; Laverty, Milliken, Milford, & Cregan, 2012). At secondary school level where students have few background understandings to leverage off and only a single course to take, an eclectic but scaled down overview course was envisioned as most beneficial and plausible.

In 2000 the modern microcontroller with flash memory had replaced the previous mid-scale microprocessor system with its discrete UV-EPROM and RAM memories. This made cost-effective simple hardware realisable for schools, and commercial products were becoming available such as the Basic Stamp (Edwards, 1998). The accessibility of suitable hardware turned me towards the primary concerns of new learners, the difficulties when using separate cross compilers, debuggers and tool chains (Ming, Longxing, & Yongming, 2012). Novice learners need concise learning objects (Mayer & Moreno, 2010) that reduce cognitive load by managing the number of unfamiliar items that need to be processed at one time (Kalyuga, 2010). Hardware that met these criteria existed but development tools were also needed. Ease of use suggested an integrated development environment (IDE) rather than one consisting of discrete tools. Along

with this choice the associated aspects of cost of programming hardware and cost of the application were important. A further concern was that any application must be useable by students at home so a free, student, or non-commercial version was essential to avoid students wanting to illegally obtain software.

Through investigation and trialling the BASCOM-AVR BASIC cross-compiler was chosen. This was chosen as it has an integrated editor, compiler, programmer and simulator making it easy for novices to grasp its many aspects as unified rather than separate. It is a professional tool with many sophisticated libraries and a highly capable student version. This means that its ongoing development is likely and the investment in building a programme of learning around it would not be lost. It is a compiled rather than an interpreted BASIC, consequently it is fast and hides none of the hardware details from learners. Some consideration was given to alternative programming environments, such as block type programming (e.g. Lego Mindstorms). A traditional programming syntax was desired however that would allow students to step more easily into further programming education. More recent criticism of abstract languages such as Lego Mindstorms is that they do not motivate students to learn as their skills are not easily transferable (Guzdial, 2004).

The most significant decision however was to use BASIC rather than C; this was not made casually as C is still the most common language used for embedded system development (UBM Tech, 2013); and student familiarity with C syntax would assist their transition into the popular Java language used in many introductory computer programming courses. BASIC though was specifically developed to remove the unnecessary syntax that complicates other languages (Kelleher & Pausch, 2005) making novices' learning more straight forward.

A line by line analysis in Table 2-1 presents one of the first exercises undertaken by learners. This is the embedded systems version of the common computer program 'hello world', where instead of text on a computer monitor an LED is flashed on and off. The BASIC version of the program is analysed in terms of its separate syntactical and semantic elements, then the C version is compared to show the significant increase in the complexity and number of individual pieces of knowledge the novice learner is faced with.

'hello world' in BASIC	**'hello world' in C**
	#include <util/delay_basic.h>
Config PORTA.3 = output	DDRA |= _BV(3);
Do	While(1) {
Set PORTA.3	PORTA |= _BV(3);
Wait(1)	_delay_ms(1000);
Reset PORTA.3	PORTA &= ~_BV(3);
Wait(1)	_delay_ms(1000);
Loop	}
Syntax to become accustomed to Do-Loop Making a pin high: *Set PORTA.3* Making a pin low: *Reset PORTA.3*	**Syntax to become accustomed to** While(1) {} Making a pin high: *PORTB |= _BV(3);* Making a pin low: *PORTA &= ~_BV(3);* assignment operators: &= |= when to use a semicolon when not to use a semicolon
Underpinning semantics Understanding of ports and pins Configuration of a pin as either input or output Infinite looping ON versus OFF effects for outputs Delay to slow a microcontroller down so that what it is doing can be 'seen' Function calls	**Additional underpinning semantics of C** Use of macros Combination of multiple operations Read, change and write back sequence Libraries of functions binary arithmetic: OR, AND, NOT milliseconds notation

Table 2-1: Comparison of syntax and semantics for Basic and C for the "Hello World" program.

This breakdown shows the range of syntax and semantic understandings students need to develop to accurately comprehend what is happening in the program. These are needed in addition to the edit, compile and upload processing concepts needed when using an IDE, and the hardware aspects of driving an LED. Minimising the number of new concepts to address with a novice at one time reduces their cognitive load and increases their chance of picking up new material (Kester, Paas, & Van Merriënboer, 2010; Mayer & Moreno, 2003, 2010; Moreno & Park, 2010).

Of note here is the command to change the state of a single pin, in BASIC it is '**Reset PORTA.1'** compared to the command used in C, '**PORTA &= ~_BV(3);'**. In C this line of code requires significant reasoning because it is comprised of several sub elements (Rist, 1989), atoms (Schulte, 2008) or 'transactions' (Mayer, 1987) that the novice must contend with, most of which are obscure and semantically challenging. This code involves changing a single bit in a register by reading the register, binary arithmetic using a condensed assignment operator with an inverted unpacked macro (1 shifted left three times) and writing back to the register, a level of detail that confuses novices who are used to human language which leaves much communication implicit (du Boulay, 1986). Note that underlying this is the compiler designer's choice to deprecate the single SBI and CBI single bit commands (because not all of the AVR registers can use them). All these transactions are extraneous noise in terms of cognitive load theory (Kirschner, 2002) as they are superfluous to the central concept being learnt and block learning by presenting cognitive interference and cause cognitive overload.

Pedagogical factors such as the esoteric nature of some languages are becoming of increasing importance in language choice for new learners (Mason, Cooper, & de Raadt, 2012) as ease of learning is seen as having a crucial role in attracting and keeping computer science students in courses. A change is now taking place in New Zealand universities to reflect this with the transition from Java to Python as a first language choice, hopefully signifying the passing of the over-used phrase "don't worry about that for now, you'll learn about it later" (A. Luxton-Reilly, personal communication, 22 May 2013).

There have been further developments in embedded systems learning environments since 2000, for example the PICAXE and ARDUINO. Subsequent reviews of the learning programme however have still found no advantage in changing, primarily due to cost. Of note the PICAXE is a popular system commonly used today in schools and continuing to develop in popularity. The PICAXE system has a highly developed IDE for the BASIC language. The IDE tokenises the code to some degree and requires a propriety boot loader and interpreter inside the PIC

microcontroller (WestAust55, 2011). This system was explored with year 10 classes for two years; the costs were one aspect of rejecting it as these were considerably greater than the AVR for 350 students. The primary concern though was that it seemed a backward step in terms of preparing students to transition to further studies as it hides many of the concepts and advanced features senior students were beginning to explore in their projects.

While a few senior students design and make their own microcontroller boards, most learners use the range of AVR development boards I have developed. Modern development boards often incorporate many sophisticated features (Laverty et al., 2012) thus presenting significant difficulty for novices. Four development boards have been designed for students (see Appendix N: The context of learning in embedded systems at Mount Roskill Grammar S) and range from very simple to more complex to counter cognitive load issues for new students. The strategy of including a prototyping area with boards is in response to meeting the requirements of technology education where students do not behaviourally follow nor are they cognitively led by the teacher in their learning but follow the participatory metaphor, where they must construct their own outcome to meet the needs of the situation they are working within. This flexibility gives rise to the not uncommon situation in senior classes where an eclectic range of interfaces are in use such as: cellular modules, infrared detectors, PIR sensors, mains switching, ultrasonic sensors, network transceivers, RF transceivers, PWM motor drivers, servos and so on.

2.3.2. The longer path: building student capability

As I began to address the wide ranging learning area of technology education in the setting of embedded systems a number of concerns became clear. The first was a clear lack of the important systems thinking in embedded systems; students were unable to keep track of the individual aspects of their projects and unable to link them together cohesively. This was important as without systems thinking students were unable to develop comprehensive solutions for clients as required by the technology curriculum. Along with this there was little appreciation of planning. Project work was egocentric, all about what the student wanted even if they were making something for someone else; and there was limited understanding of the importance of stakeholders and few ideas on how to interact with them. I also saw too many students unable to progress beyond writing the most trivial of programs. As a teacher I had begun to realise the importance of what I had taken on.

2.3.3. Background to concerns with programming

In 2005 I began to look at students' issues with programming. Using experiences from industry I evaluated student understandings using a capability maturity model (Paulk, Curtis, Chrissis, & Weber, 1993); it was evident that most students were seldom progressing beyond an initial or ad-hoc level with programming. Whilst students are not expected to become experts, as this takes many years (Winslow, 1996), I felt it unsatisfactory that they should not know how to write non-trivial programs by the end of their course, a not uncommon phenomenon (Carter & Jenkins, 1999; McCracken et al., 2001; Tew, Dorn, & Schneider, 2012). A goal was set for students of being able to self-manage the development of their projects and programs for small embedded systems.

In 2006 I investigated a range of computer programming textbooks and like others (de Raadt, Watson, & Toleman, 2009a; Robins, Rountree, & Rountree, 2003; Soloway, 1985) identified them as having high levels of syntax content but with few insights into how to begin to address the lack of capability my students exhibited. These texts, like many introductory computer science courses leave students to "flounder on their own" (Engel & Roberts, 2001, p. 23). The texts specifically lack useful models on how to or when to use syntax and semantic knowledge. This is in direct contrast to technology education's key aspect of technological practice; this is not content knowledge but process (Custer, 1995) or 'know-how' knowledge (Ropohl, 1997). The difficulty when dealing with this type of knowledge is that it is not usually explicit but implicit and tacit (Compton, 2004), i.e. we don't know that we know it. As programming knowledge becomes implicit or tacit, experts, including experienced teachers, forget just how much of it they know, how they learnt it and importantly do not know how to pass it on (du Boulay, O'Shea, & Monk, 1981). The need for an explicit 'know-how' methodology is highlighted in literature as highly important for novice learners (Robins et al., 2003; Rogalski & Samurcay, 1990; Winslow, 1996) though there is limited guidance on how to develop it.

Initial trials of a methodology began with flowcharts (Crews, 2001) and the analogous Nassi–Shneiderman diagrams (Berendsen, Krammer, & Onderwijskunde, 1992). These proved to be a false start. Whilst they could diagram aspects of sequences and algorithms "flowcharts have been a failure" (Harel & Gordon-Kiwkowitz, 2009, p. 89) as serious software development tools as they are unable to express the complex set of behaviours and reactions of embedded systems (Harel, 1987). In 2007 I began exploring the use of state machines, and while they seemed

promising a number of issues had to be addressed such as: how to introduce the new ideas and how to capture the methodology or process of converting diagrams to program code.

Investigations of several techniques for turning state diagrams into program code and trials with applications led to the use of UMLPad (Bignami, 2013), a free UML diagramming tool. This application was chosen as its output files are machine readable XML, which made it easy to automate the generation of program code and thus turn students' focus away from syntax to problem solving about their application. I chose a simple while-loop pattern that was easily understandable for students (and easy to follow as I often assist in debugging their code). I wrote a program called State Charter that parsed the XML file turning it into a template of BASIC or C code that could be compiled directly by Bascom-AVR or GCC. Students still had to write the detail into their code, however two significant challenges of learning to program were being addressed: the confusion around syntax that challenges novices, and the higher level challenge of turning problems into programs (Coull & Duncan, 2011; Lahtinen, Ala-Mutka, & Järvinen, 2005). The teaching approach I have developed for learning about state machines is documented in the course textbook (Collis, 2013, pp. 478–525). The growth in size, scope and accuracy of student programs at the senior level was immediate along with a further benefit of state machines (and a significant requirement of technological practice), the ability to directly engage end-users in software design (Harel, 2009).

After that initial success I began in 2008 to develop the current application 'System Designer'. This not only incorporates a student-friendly state machine diagramming tool with automatic generation of program code, but a suite of other tools for students as well. My goal was to present a cohesive approach to learning in embedded systems and to develop a tool that reflected the integrated nature of its many facets; so I quickly began to replace the eclectic range of diagramming tools I had assembled for students into a single application.

When using state machines students immediately began progressing beyond the localised thinking issues of novices (Letovsky & Soloway, 1986) with more abstract reasoning about clients' needs rather than program issues. Coupled with classroom success are continued high levels of achievement in NCEA, New Zealand Scholarship, national competitions and international Science and Technology Fairs.

My observations during classroom activity however correspond with other findings that performance is not always a sound indication of understanding (Ben-Ari, 1998). Many students (including those that had taken the course for several years) were unable to identify where in

program code a fault lies. Often students will rewrite code unnecessarily breaking correctly functioning programs and some students will even change hardware when a program does not work in an effort to fix a problem. My concern with capability is that echoed by many other educators (Getty, 2009; Goris & Dyrenfurth, 2010; Ramsden et al., 1993; Rowe & Smaill, 2007), that students are able to gain good grades but still carry significant misconceptions about core concepts with them into further learning.

2.3.4. Levels of programming understanding

Questioning in class revealed that many students (both novices and even the more experienced) would often refer to textual or surface descriptions of their program code rather than the deeper functional aspects of their programs. Functional thinking is a core progression required for students in technology (Ministry of Education, 2010) and operating at the level of what something is rather than what something does indicates a very low level of technological understanding (Compton & Compton, 2011). In computer science education Schulte (2008) identified the learning gap between textual and functional understandings of code as the key concern for novice programmers.

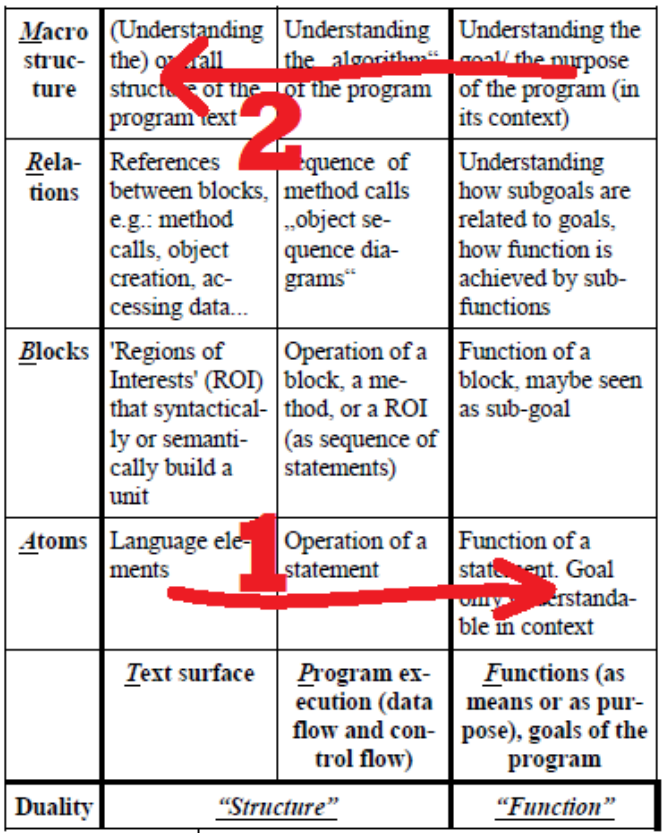
A number of researchers have explored understandings in programming thinking, categorising students over a range of five levels from surface to deep approaches to programming (Booth, 1997; Bruce et al., 2006; Eckerdal & Berglund, 2005; Linn & Dalbey, 1985). When surface thinking students see programs at two distinct levels: at the most shallow level of pure text or as lines of program code that create actions. As students progress into deeper understandings they move through three more levels: they have a complete overview of the program, they appreciate that programming is problem solving thinking and ultimately students see computer programming as an expression of their thinking. Thuné & Eckerdal (2009) stress the importance to teachers of not leaving students with surface (textual) understandings but creating sufficient variation in the deeper dimensions, thus allowing students to abstract the other levels of understanding. This is especially important when helping students master action thinking, a difficult “almost mystical way of thinking” (Thuné & Eckerdal, 2009, p. 342).

A second model of program comprehension (Schulte, Clear, Taherkhani, Busjahn, & Paterson, 2010; Schulte, 2008) presents a student’s view of programming not as one sequential progression but as a number of separate progressions from structural to functional understanding and across separate sub domains of understanding in programming as in Table 2-2.

Macro structure	(Understanding the) overall structure of the program text	Understanding the „algorithm" of the program	Understanding the goal/ the purpose of the program (in its context)
Relations	References between blocks, e.g.: method calls, object creation, accessing data...	sequence of method calls „object sequence diagrams"	Understanding how subgoals are related to goals, how function is achieved by subfunctions
Blocks	'Regions of Interests' (ROI) that syntactically or semantically build a unit	Operation of a block, a method, or a ROI (as sequence of statements)	Function of a block, maybe seen as sub-goal
Atoms	Language elements	Operation of a statement	Function of a statement. Goal only understandable in context
	Text surface	**Program execution (data flow and control flow)**	**Functions (as means or as purpose), goals of the program**
Duality	***"Structure"***		***"Function"***

Table 2-2: Block model of program comprehension (Schulte, 2008)

Schulte presents the idea that learning is chaotic and does not always need to follow a usual progression for example from bottom-up in the table and from structure to function, but that understanding will develop in different blocks at different times. He recommends that not all blocks need to be taught in a course and different paths can be taken through the model.



Macro structure	(Understanding the) overall structure of the program text	Understanding the „algorithm“ of the program	Understanding the goal/ the purpose of the program (in its context)
Relations	References between blocks, e.g.: method calls, object creation, accessing data...	Sequence of method calls „object sequence diagrams“	Understanding how subgoals are related to goals, how function is achieved by sub-functions
Blocks	'Regions of Interests' (ROI) that syntactically or semantically build a unit	Operation of a block, a method, or a ROI (as sequence of statements)	Function of a block, maybe seen as sub-goal
Atoms	Language elements	Operation of a statement	Function of a statement. Goal only understandable in context
	Text surface	**Program execution (data flow and control flow)**	**Functions (as means or as purpose), goals of the program**
Duality	*"Structure"*		*"Function"*

Figure 2.3: MRGS learning flow in programming

Reviewing what has been happening in class with regard to this model, the path taken is initially from the bottom left and then from the top right backwards as in Figure 2.3. The model provides a powerful personal insight in that the gaps in student understandings experienced in class relate to the blocks in the centre regions of the model.

2.3.5. Conceptual understandings

Due to the hidden or black box nature of electronics and embedded systems software, deep conceptual understandings are challenging for students to develop, yet it is these deep understandings that make such a difference to student outcomes (Hattie, 2011). It is important then to define what conceptual understandings are. McCormick (1997, p. 143) defines them as "relationships among items of knowledge" clearly separating them from factual knowledge. This aligns with research that misconceptions are caused by failure to link understandings (Clancy, 2004)

Much research has been carried out investigating exactly what concepts students should know in computer science and inventories have been developed through analysis of textbooks and by consulting with experts to identify core concepts (Goldman et al., 2008, 2010; Tew & Guzdial, 2010; Zendler & Spannagel, 2008). Other inventory work in computer science includes

conceptual reasoning (Krone, Hollingsworth, Sitaraman, & Hallstrom, 2010). In reviewing literature on conceptual understandings there is a common agreement on many of the things that novices need to know and struggle with such as program flow, sequence, variables and assignment.

An important understanding that is emerging in literature is that of concepts that act as gate keepers for student. There is growing research in a number of subject areas to identify threshold concepts (Land, Cousin, Meyer, & Davies, 2005). These are seen as representing either blocks to or major leaps in understandings (Bradbeer, 2006) and lead to fragile knowledge if not dealt with (McCartney et al., 2009). In computer science the object oriented concept and pointers have been identified as potential threshold concepts (Boustedt et al., 2007). Further research into threshold concepts is still seeking to quantify the idea of threshold in computer science education, both in how it relates to teachers' ability to identify students' misconceptions (Shinners-Kennedy & Fincher, 2013) and that skills may be thresholds as they are required to implement understandings (Sanders et al., 2012).

Threshold concepts have been used to support the development of anchor concepts and their structure into an anchor graph (Mead et al., 2006) as in Figure 2.4. The ordering of concepts provides an insight for educators into how to reduce cognitive load for novices by knowing which concepts to begin with (called anchors) and then working onwards from those understandings with new concepts.

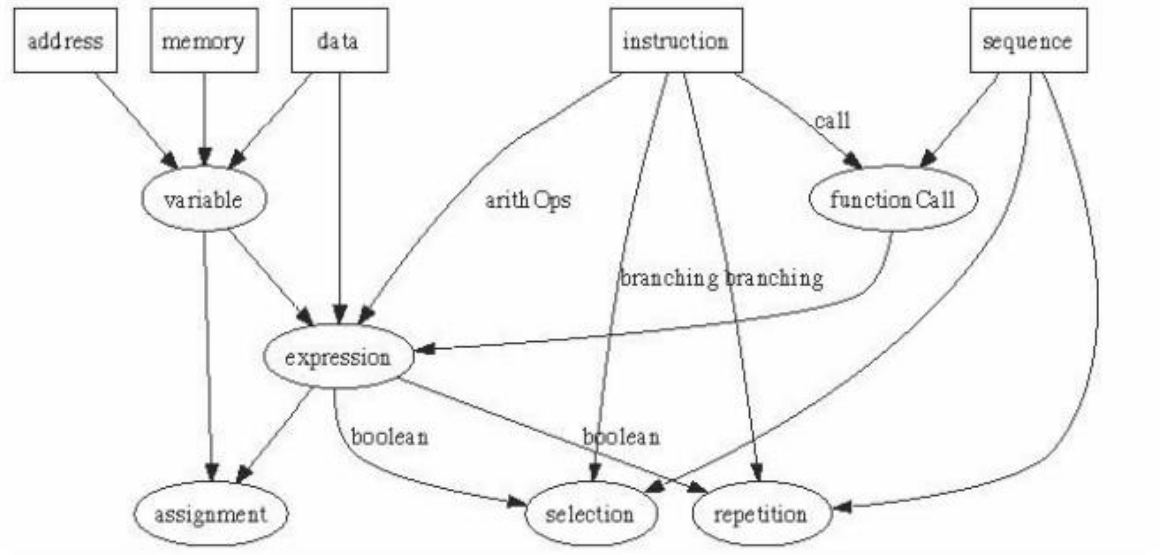


Figure 2.4: Directed graph on anchor concepts (Mead et al., 2006)

2.3.6. Relational understanding

The directed graph of concepts in Figure 2.4 and Schulte's model in Table 2-2 both highlight that conceptual understanding is an understanding of relationship. For instance a variable is an understanding not of its own but one of relationship between the functions of memory, address and data.

Two prominent models of learning echo the relevance of relationship to capability. The Solo taxonomy (Biggs & Collis, 1982) presents five levels of understanding based upon the ability to connect information, and the revised Blooms taxonomy (Krathwohl, 2002) has six levels. Beginning with the second level in both models is the important idea of connection of information with other information. Making connections between ideas stresses the importance of moving from the idea of what something is to what something does. The priority for conceptual understanding must not be the understanding of isolated pieces of 'know-what' knowledge but an ability to form relationships between them so as to be able to select which ones should be used and how to bring them together in patterns. This highlights the crucial role of tacit 'know-how' knowledge, knowing how things work and how things function together.

In embedded systems the relatedness of understandings is expressed fully in its name, 'systems', a system is more than collections of devices or circuits but interrelated processes that work together to transform information or objects. Without linked understandings learners cannot begin to comprehend the full nature of programming or embedded systems.

2.3.7. Notional machine

Research into conceptual understandings of experts shows their ability to form a mental representation or model of a program's function (Gott, 1988; Schulte et al., 2010; Vessey, 1985). A mental model is an internal memory construct that represents how the elements of something are in relation to one another. A mental model gives programmers the ability to capture the working of a program (Gott, 1988; Kurland, Pea, Clement, & Mawby, 1986) and hence accurately predict its operation (Ben-Ari, 1998). Inadequate and faulty mental models result in misconceptions (Ben-Ari, 1998) and are clearly linked to poor programming ability (Johnson-Laird, 2006; Mosemann & Wiedenbeck, 2001; Perkins, Schwartz, & Simmons, 1988; Samurcay, 1989; Schulte, 2008). There is a crucial need for students to develop dynamic and simple unsophisticated mental models (Abrantes et al., 2008; du Boulay, 1986; Guzdial, 2004). As educators it is significant that we realise that people build their own mental models without

prompting, and without explicit instruction these can be highly inaccurate (du Boulay, 1986; Schulte, 2008).

In computer science this crucial understanding of mental models has been captured by the need for learners to develop the concept of a 'notional machine' (du Boulay et al., 1981). This is a two-step process which includes the development of how components function and their relational understanding of "how change in one part of the system causes a change in another part" (Mayer & Moreno, 2003, p. 47). These patterns of relationship are called 'schemas' or 'plan schemas' (Rist, 1989) and come about in the mind as students develop their own structure, organisation, rules and relations for subject matter (Dreyfus & Dreyfus, 1980; Glaser, 1984; Medin, Rips, & Smith, 2012). It is the development of a notional machine (i.e. it supports relational understandings) that should become the dominant activity for novice learners (Kalyuga, 2010; Soloway, 1985) however it is not given the same level of emphasis by textbooks or educators as notation, structure and pragmatics (Schulte & Bennedsen, 2006).

The concept of a notional machine applied to embedded systems then must be able to not only support conceptual understandings of programming but also conceptual embedded systems understandings. These are the understandings of the interrelationship between hardware and software, reactivity (the interrelationship between system and environment), state (the interrelationships between what has happened, what is now happening, and what will happen), concurrency (the interrelationship between the system and time) and also validation (the relationship between system and context). Each understanding is much more than factual knowledge but a working understanding or mental model. While these understandings are well understood in literature there is further opportunity for their exploration both as understandings in their own right and how they align with understandings of learning to program.

2.4. Summary

In this chapter three aspects of embedded systems have been covered, what they are, why they should be taught and what should be taught. Embedded systems are everywhere; they silently manage our environment for us in a vast number of ways without us even being aware they are doing it. This is a significant appreciation that students develop through learning in embedded systems as it provides a sound platform for becoming 'literate' in technology and supports learners' development as knowledgeable citizens of the future. The understandings students require are not trivial though, and while research into the important aspects of conceptual

relational understandings in learning to program (mental models and notional machines) are growing, the textbooks and beginning courses in programming still appear highly focussed on surface or textual understandings. The important conceptual understandings in the area of embedded systems learning are well covered in literature however the opportunity exists to explore the unique conceptual understandings in this area further.

Chapter 3. How best can embedded systems be learnt?

There are many facets to learning and the practice of teaching. In Hattie's work examining over 900 meta-analyses (encompassing 60,000 studies of over 240 million students) he found that almost "everything works" (Hattie, 2011, p. 2) but that some practices have significant benefits over others and are more viable for teachers to implement. This chapter will introduce some of the key findings and relate them to the metaphors of acquisition (behaviourism through to constructivism) and participation (socio-cultural practice) from Section 2.2, as well as the current teaching practices highlighted in engineering educational literature and practical aspects of learning in embedded systems.

3.1. What works best

A significant result from Hattie's analysis is that setting challenging tasks has one of the largest effect sizes on student outcomes (p. 29); in this way the learning of embedded systems (because of its demanding nature) is primed for success. Learning for deep understandings rather than surface learning is another key contributor to student outcomes (Hattie, 2011) but not easily achieved (Case & Gunstone, 2002). The concerns identified and discussed around mental models and conceptual understandings in section 2.3.2 align clearly with a focus on deep rather than surface learning. In contrast, behavioural objectives however are poor improvers of student performance.

Of all the teaching strategies the immediacy and amount of formative feedback (Alton-Lee, 2003; Hattie & Jaeger, 1998; Hattie & Timperley, 2007; Hattie, 2011) is one of the most significant contributors to student outcomes, and one which teachers have significant control over. Formative feedback should be focussed on one of three aspects of students' performance: at the task level (right or wrong) for novices, at the process or strategies level for proficient learners or at the metacognitive level for competent learners (Hattie, 2011). The opportunity for students to gain frequent feedback is however a concern in embedded systems learning because much of the technology is 'black box'; that is, it is hidden from view and must be inferred from system behaviour. This is explored later in this chapter.

3.2. Deductive and inductive teaching

Two common approaches to teaching in engineering literature are described as deductive and inductive (Caspi et al., 2005; Felder, 2012; Prince & Felder, 2006). These are two methods involving the cognitive process of reasoning or thinking so as to form conceptual understanding (Sternberg, 2010). Deductive can be seen as top-down where principles and theory are introduced first and supported by specific examples to strengthen understanding; while inductive learning is bottom-up where specific experiences are given first and structured by the teacher in a way so as to produce generalisations or abstract understanding (Arends & Kilcher, 2010).

3.2.1. Deductive teaching

When teaching using a deductive process, theory and principles are initially presented out of context (Anderson et al., 1995); the student is required to learn to apply these in later practical or contextualised situations. Teaching deductively is maligned in some engineering literature for surface patterns of understanding either via its rote learning of information or routine application of formulae (Felder & Brent, 2004a, 2004b). This deductive learning approach is widespread practice in engineering and science (Prince & Felder, 2006); sometimes being criticised as "content tyranny" (Prince, 2004, p. 229) and has a focus on extrinsic rather than intrinsic motivation (Felder & Brent, 2004b).

The approaches and outcomes described by researchers though, do not seem to correlate with a cognitive deductive reasoning model of learning, but suggest behaviourist practices instead; just as they are still encountered in many school classrooms (Thornburg, 1994), and factor heavily in the practical lessons of technical teachers (Harwood & Compton, 2007). In my own class the outcomes of behaviourist instruction are often encountered when a student asks "what do I do next?" On occasion I have fielded even more strongly delivered requests: "just tell me what to do", "I don't want to think", or even "but that's your job, you're the teacher". These students are satisfied with low level task work and do not want to progress into process or strategic thinking. There is a dichotomy here in that students prefer it (Felder & Silverman, 1988) and are comfortable with it (Yadav, Subedi, Lundeberg, & Bunting, 2011) yet acknowledge that their interest in subjects is often reduced because of it (Tseng, Chang, Lou, & Chen, 2011).

Just as behaviourist approaches cannot be paralleled with only formal classroom/lecture approaches it is important not to parallel formal classroom teaching or lectures with deductive

teaching approaches. These things do not always align, as lectures may incorporate inductive methods of thinking as well, just as practical lessons can be taught deductively.

3.2.2. Inductive teaching

In inductive teaching instead of being given rules, students are given tasks, projects or problems and through these activities abstract (develop a sense of) the rules for themselves. This is described as a constructivist approach (Arends & Kilcher, 2010; Felder & Silverman, 1988; Felder, 2012); a constructivist theory of learning shifts the emphasis away from the centrality of the teacher towards the student (Barak, 2009; Prince & Felder, 2006). In engineering it has been shown to produce significantly deeper understandings of content, increases in student motivation and interest in their work, improved design skills, intellectual growth and increases in content knowledge (Adamo, Guturu, & Varanasi, 2009; Bertels et al., 2009; Felder & Brent, 2004b; Felder, 2012; Mantri, Dutt, Gupta, & Chitkara, 2009; Mioduser & Betzer, 2007). Inductive or constructivist learning however requires a shift in focus for both teachers and students. Assignments and exercises that cover content and which are centred on getting the 'correct answer' are replaced by the tools of inquiry, project and problem based challenges (Felder & Brent, 2004b; Harkin, Callaghan, McGinnity, & Maguire, 2002), which require a gradual and careful withdrawal of teacher support (Prince & Felder, 2006).

One underpinning to this approach to learning is the powerful pedagogical principle of frequent formative feedback. Feedback is ever present in a problem solving environment, as rather than having to wait for some external feedback mechanism to come into play, such as a test or exam that will check if a student understands or not, the student receives it immediately through their work. A further crucial aspect of education is giving students sufficient time and opportunity to learn (Alton-Lee, 2003); Case and Gunstone reduced their course content by 25% so that they could "cover less, uncover more" (2002, p. 463). The reduced content and increased time allocation of inductive/constructivist learning has led to its criticism compared to a deductive approach (Yadav et al., 2011).

While inductive teaching is aligned with active methods such as inquiry-based learning, project-based learning and problem-based learning it is important not to parallel these as inductive teaching may just as easily feature in formal classroom and lecture situations as part of direct instruction or demonstration (Eggen & Kauchak, 2006).

3.2.3. Deductive or inductive?

The discussion around inductive and deductive methods is an ongoing one for many educators; none less so than in second-language learning (Fischer, 1979). In second language learning a simplification has often been applied that an inductive approach may be more suitable for simple language structures and deductive learning for more complex ones. This is questioned though as all situations ultimately present a complex range of interrelated variables around the individual, the teacher and the material (Decoo, 1996). Whilst the practices of deductive methods in engineering literature are commented on negatively compared to inductive methods, the challenge may be to distinguish the practices which are behaviourist as opposed to cognitive, and then see that inductive and deductive are not "two opposing monolithic techniques" (Worster, 2013, p. 1). This leaves educators at liberty to choose the most appropriate strategy (e.g. direct instruction, inquiry, project or problem based learning) for the goals of the learning context, the student, the teacher, the material to be understood and the resources available.

3.3. Students as participatory learners

Learning as participation is the new paradigm in education and at the centre of technology education in New Zealand. This is a cognitive apprenticeship approach where implicit knowledge is made explicit or modelled in authentic contexts by experts (Brown, Collins, & Duguid, 1989; Lave & Wenger, 1991). This extends problem based learning, where students are involved in "contexts similar to the ones they will face in the future professions" (Williams, Iglesias, & Barak, 2007), to involvement in a community working on solving actual problems. An example of this is open-source software development (Ben-Ari, 2004). When students participate in authentic practice the teacher no longer sets the learning objectives which are instead negotiated as the situation unfolds and student learning needs are recognised (Compton & Harwood, 2003). In this approach to learning teachers no longer hold the power of assessment over students; the student measures him/her-self against the needs of an authentic situation and real stakeholders.

As teachers replace behaviourist with cognitive practices and cognitive with constructivist practises their roles become increasingly more difficult (Blumenfeld et al., 1991). It is more challenging again for the teacher in a socio-cultural learning environment as each student needs to engage with an authentic context. A teacher's ability to negotiate a participatory learning pathway with a student is not straight forward as it requires considerable experience to observe and interpret misunderstandings and to provide just-in-time explanations, support and guidance.

It necessitates an ability to see clearly how students are progressing within what often appears as a chaotic working environment (Catalano & Catalano, 1997). The student must also be willing and ready to operate at the highest learning levels of self-regulation and metacognition; and the teacher must be able to provide them with feedback at this deep level (Hattie, 2011). The key to this is a teacher's pedagogical content knowledge, a rich blend of subject matter and understandings of educational practice (Shulman, 1986).

Whilst the participatory model is at the forefront of pedagogy in technology education in New Zealand, as a general method it has been criticised in that finding authentic situations for learners is both inefficient for learners and places unreasonable expectations upon educators (Ben-Ari, 2004). The current programme for year 13 students (their final year of secondary school technology) is a fully participatory model with students working on unique projects with individual clients; it is highly demanding on staff. In science education Ben-Ari comments that enforcing pure principles of participatory learning would be an "appalling waste of time of highly trained specialists" (Ben-Ari, 2005, p. 374) and it should be limited to only analysing real communities of practice. These concerns parallel the more recent changes in technology education in New Zealand as it moves back from its purist approach with the reintroduction of a non-participatory model of knowledge development.

3.4. Use of real hardware in embedded systems learning

The use of real hardware is the only medium in which the learner can fully appreciate the "very real limitations and quirks" (Koopman et al., 2005, p. 524) of embedded systems because of their reactive and real-time nature. Laboratory work also prepares students for the real world and is crucial in developing students into experimenters (Ernst, 1983). Exposure to and use of hardware though does have significant limitations. It is resource hungry in terms of time, space, hardware and staffing but its greatest limitation is that it does not reveal a great deal about what is happening inside an embedded system (Cho, 2009) and thus cannot support students in building viable mental models.

Embedded system development itself is a highly complicated process and novice understandings are stretched across a wide range of pieces of equipment and interactions: editor, compiler, programming software, programming hardware, sensors, actuators, microcontroller, compiled program and user interfaces. As depicted in Figure 3.1 most of these are hidden from the

learner, who will write code into an editor and input information via a user interface but can only see an actuator or a user interface response.

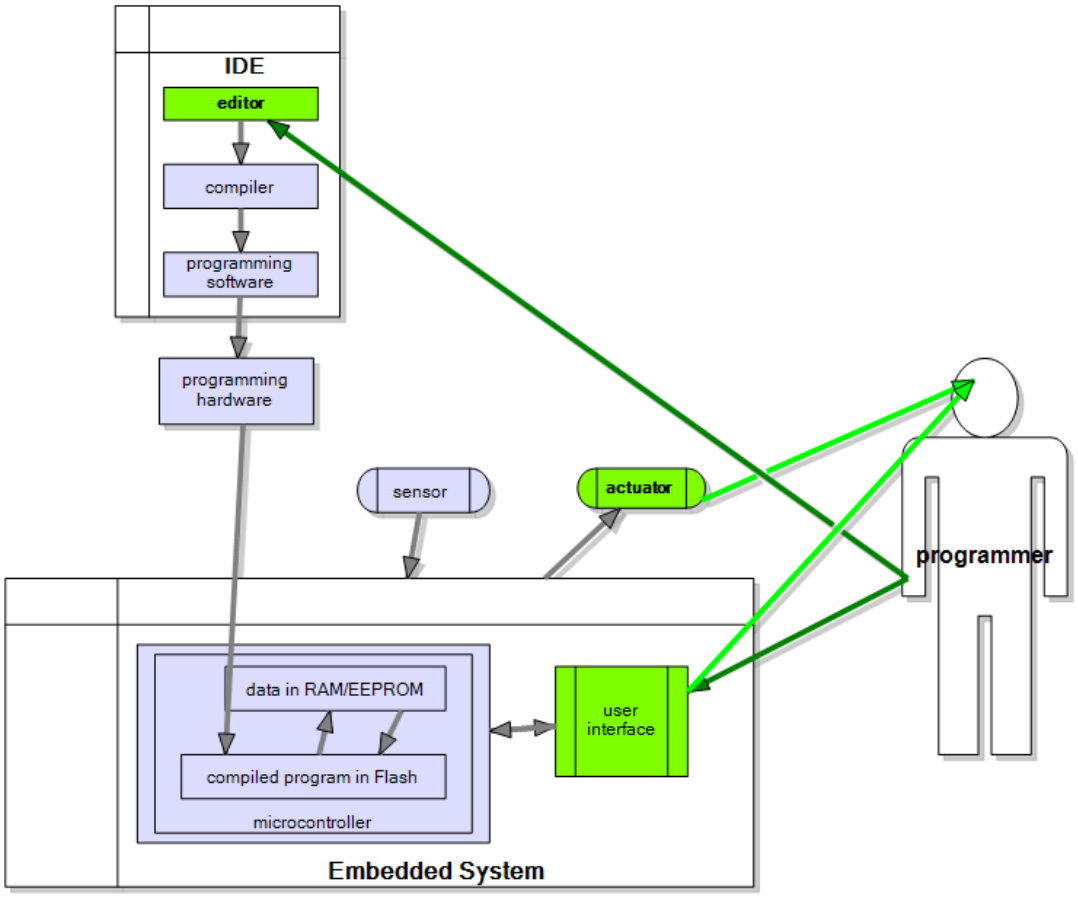


Figure 3.1: Components and interactions in embedded system development

A novice student must infer the many links and processes without any prerequisite knowledge; in learning to program this has been likened to "learning to use a toy construction set … but as if inside a darkened room" (du Boulay, 1986, p. 58). With feedback being such a crucial aspect of learning, Figure 3.1 reveals just how little feedback there is for the learner about the hidden elements of an embedded system. In a situation like this the limited feedback can have negative implications for a learner threatening how they see their own capability (Hattie & Timperley, 2007).

In seeking to identify school students' conceptual understandings I carried out an exercise to expose their conceptual understandings or mental models by having them complete conceptual diagrams and descriptions (Perkins, 1991) of their embedded systems project work. This involved ten students of electronics; eight were year 10 students who had taken a half year course and two were year 13 students who had taken over three years of electronics courses. Each was given a diagram (Figure 3.2) with instructions to write as full a description as possible of their

system. Two of the year 10 students limited their response to drawing smiley faces on the programmers; however the remaining six students adequately described the processes undertaken in programming a system. None of the six however described the internals of the embedded system at all. The two year 13 students described deeper and correct understandings about the embedded system itself (see Figure 3.3 for one of the responses).

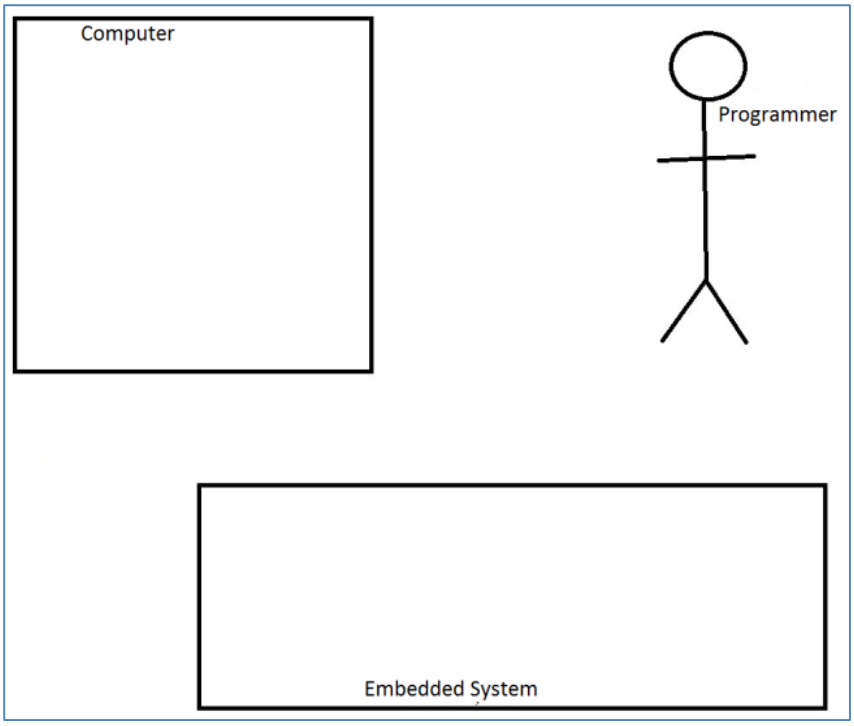


Figure 3.2: Conceptual model template for embedded systems development environment

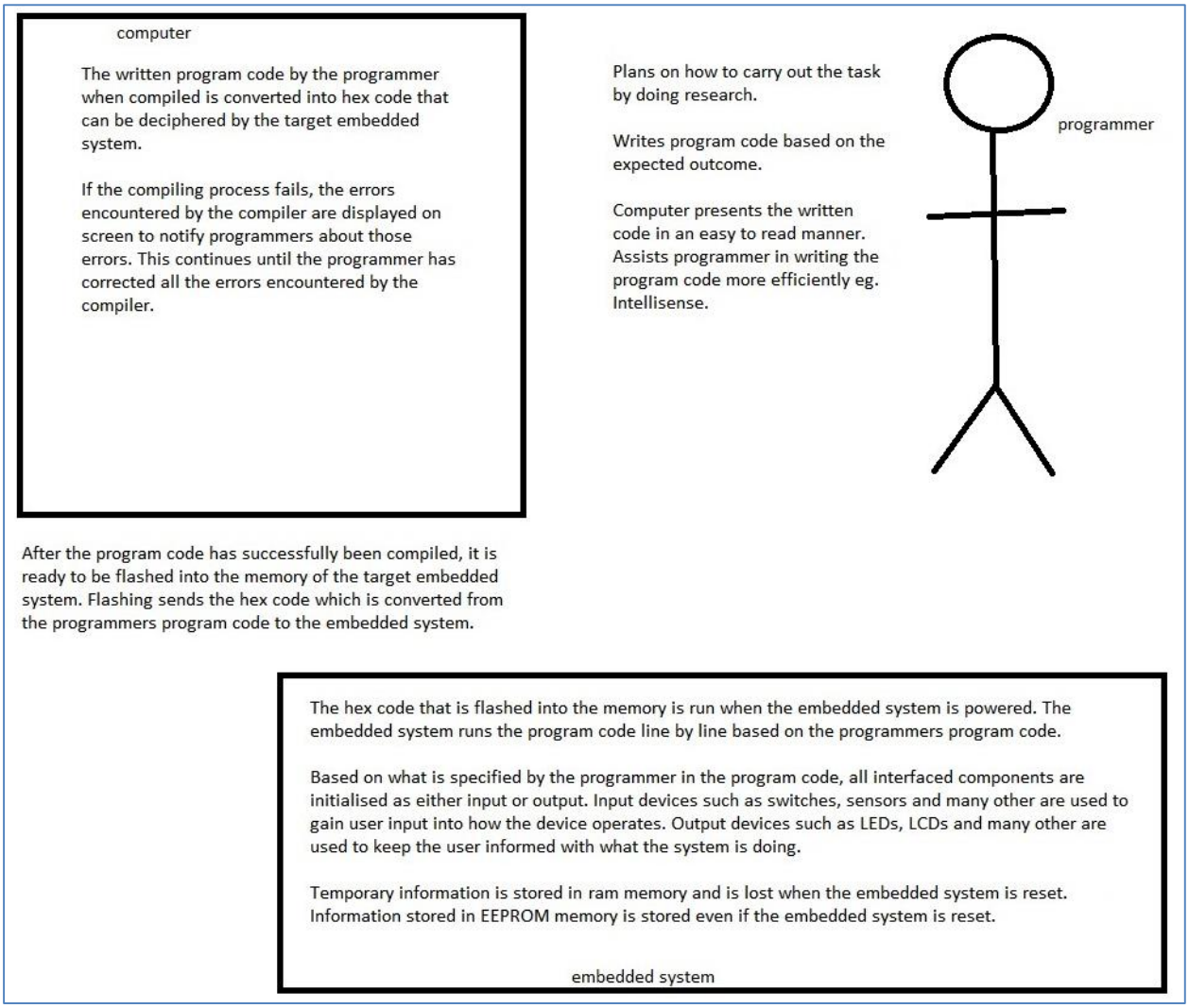


Figure 3.3: Year13 student conceptual response

These diagram responses highlight the concerns that embedded system development is a hidden and complex process and that novice students have no concept of what is happening inside an embedded system, even though they can write a simple program for one. As du Boulay wrote "it takes a long time to learn the relation between a program on a page and the mechanism it describes" (1986, p. 59).

3.5. Conceptual models and visualisation as learning tools

An approach used to develop conceptual understandings or mental models is to use conceptual models; these are the "words and diagrams that are intended to help learners build mental models" (Mayer, 1989, p. 43). For novice programmers the results of dynamic conceptual models or visualisations are highly positive, and testing has revealed key aspects of: increased reasoning and planning (Baldwin & Kuljis, 2004); improved models of computer memory (Kaczmarczyk, Petrick, East, & Herman, 2010) and value assignment (Ma, Ferguson, Roper, &

Wood, 2007). Importantly visualisation supports students moving from syntactic or surface features of software to semantically oriented understandings (CAÑAsf, Bajo, & Gonzalvo, 1994). One of the key aspects of visualisation is the opportunity to control extraneous aspects of a situation and thus reduce cognitive load for learners by allowing the student to focus on only the essential points (Mayer & Moreno, 2003, 2010).

A number of visual strategies for learners exist in programming; this study is focussed on software visualisation which is distinct from visual programming languages such as Lego Mindstorms, Scratch and Alice that use iconic syntax rather than written languages. Software visualisation is further broken into two sub-categories: algorithm visualisation and program visualisation. Algorithm visualisation often includes iconic elements such as in flowcharts and is used to support understanding of complex algorithms such as sorting. Program visualisation simulates the running of a text based program. In embedded systems program visualisation may often be found in debuggers and simulators, though this is generally at a low level via compiled assembly language rather than using a high level language view of the program.

3.5.1. The key aspect of using visualisation

For teaching, simply developing and using modelling and visualisation tools will not produce the desired pedagogical results; this is akin to behavioural training. Students only learn when actively engaged in the learning process (Coll, Taylor, & France, 2005; Hanke, 2008; Rajala, Salakoski, Laakso, & Kaila, 2009) as the visualisation does not hold the learning itself but serves as a catalyst for deep understanding to take place (Ball & Eick, 1996). When a teacher uses a visualisation the students also need training and must be explicitly helped to understand and interpret its graphical elements and processes (Ben-Ari et al., 2011).

More importantly students and teachers have two different views of models. Teachers are experts who understand the limitations of the analogy or metaphor that a model represents and what can be taken from the model whereas students are more likely to see the model as true or real and learn the model rather than the concept (Coll et al., 2005; Gilbert, 2004).

Lastly while many teachers regularly review and alter their programmes of learning, the addition of such a significant new tool as a visualiser takes considerable time and energy on their part (Naps et al., 2003); and while teachers believe that visualisation is beneficial they are mixed in their use of it as a strategy. Teachers feel visualisers can distract from their existing programs of learning (Hundhausen, Douglas, & Stasko, 2002) or even displace essential aspects of their

already busy curricula. A key aspect of supporting teachers is the need for high quality support materials for a visualiser (Levy & Ben-Ari, 2007). Pedagogical support materials are discussed in detail in Chapter 6.

3.6. Summary

Embedded systems are complex, with many unfamiliar concepts for learners; and the hidden nature of these systems makes the concepts difficult for learners to access. Computer programming and embedded systems are new paradigms for students fraught with misunderstanding created by this hidden black box nature. As educators our goal is not the regurgitation of facts but the advancement of students to progressively deeper, sometimes mystical, understandings. This can be constrained by the lack of readiness of some learners to engage in tasks that build learning capability as they reject the effort required for the learning.

Powerful theory and tools exist to help us as educators understand and support student learning. These include behavioural, cognitive top-down, constructivist bottom-up and participatory educational theories to provide insight into the development of understanding. Attached to these is a vast repertoire of strategies to choose from, such as direct instruction, inquiry, project and problem based learning. Engagement with this breadth of strategies is challenging for some teachers who prefer single, particularly behavioural, modes of teaching.

The development of viable mental models (conceptual understanding) using dynamic conceptual models such as computer-based visualisations offers tremendous support for learning but only when used in conjunction with real world contexts that facilitate the deepest levels of understandings. While the effectiveness of visualisation tools is positive the ancillary aspects involved in their development are just as important, as without quality resources, intuitive modes of operation and application flexibility, they will not be effectively integrated into programmes of learning.

Chapter 4. Classroom action research on visualisation

These trials were undertaken over a period of three years in my classroom. While they present anecdotal results of visualisation and it would be inappropriate to generalise detailed results from them, they have been highly useful in stimulating the effort to undertake this research and guiding the direction of it.

The simulator built into the Bascom AVR IDE was trialled with students as part of classroom practice. Observations and some student comments were collected as part of action research; the general findings from the exercises were that it did not support student learning.

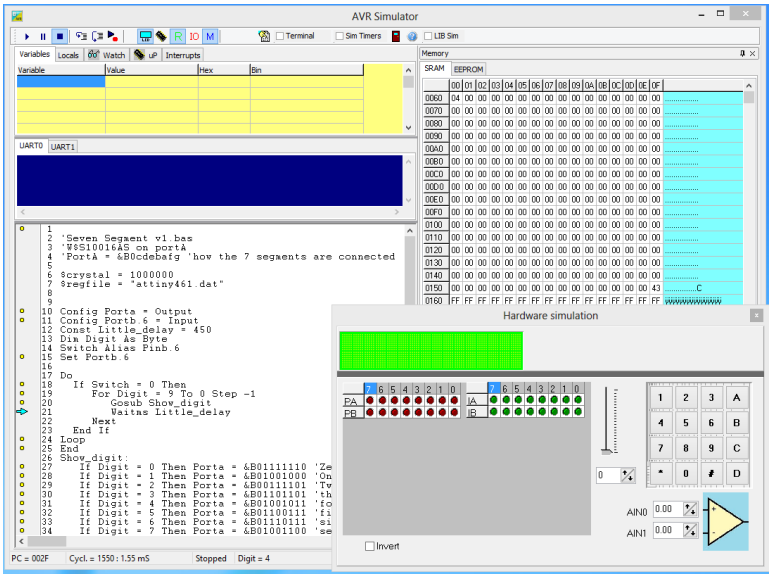


Figure 4.1: BASCOM-AVR simulator

Two tasks were developed and run twice with two different year levels. In the first task new learners in year 10 used the tool to simulate a simple program and monitor the ports (red and green LED type objects in Figure 4.1) to depict a running LED sequence. Student comments indicated they found the simulator overwhelming, and that they could not see the parallels between the images in the simulator and their own projects. The second set of exercises was developed for students who were in their second course in Electronics at year 11, and made use of the analog to digital conversion (ADC) and liquid crystal display (LCD) simulation. The exercise was aimed at helping students see the process of analog to digital conversion. The trial found some small success with students; these students had not used the simulator previously but were overall more familiar with microcontroller programming and did not struggle as much with the unfamiliarity of the detail in the simulator. The students did not have enough abstract

understanding of the process to complete their own practical exercises with Voltage dividers, LDRs and thermistors. Student comments about the value of the exercise were that it did not help. The final analysis was that the simulator did not represent a suitable analogy for the students at their level of understanding.

Whilst these inquiries form part of my own classroom observations and reflections and are thus highly contextualised, they yielded anecdotal results about the complexity of the simulator and its distance from reality for my students, and thus led me to develop 18 simpler visualisations which have all been used with students. Each visualisation was developed to directly model a single aspect that would allow students to see the interrelationships between the various entities and then use their new understanding in their own work. All these visualisations were commented on favourably by students.

The visualisations do not represent full models of programs and hardware but capture single aspects of the software-hardware interaction such as: simple output control, analog to digital conversion, and hardware timers. Figure 4.2 shows a visualisation to support students within the introductory course when they move from individual LEDs to a seven segment display. In one class, 18 of the 28 students initially struggled to successfully program their displays to show desired patterns. After using the visualisation with those who were struggling 16 were able to successfully program their own hardware without resorting to further use of the visualisation.

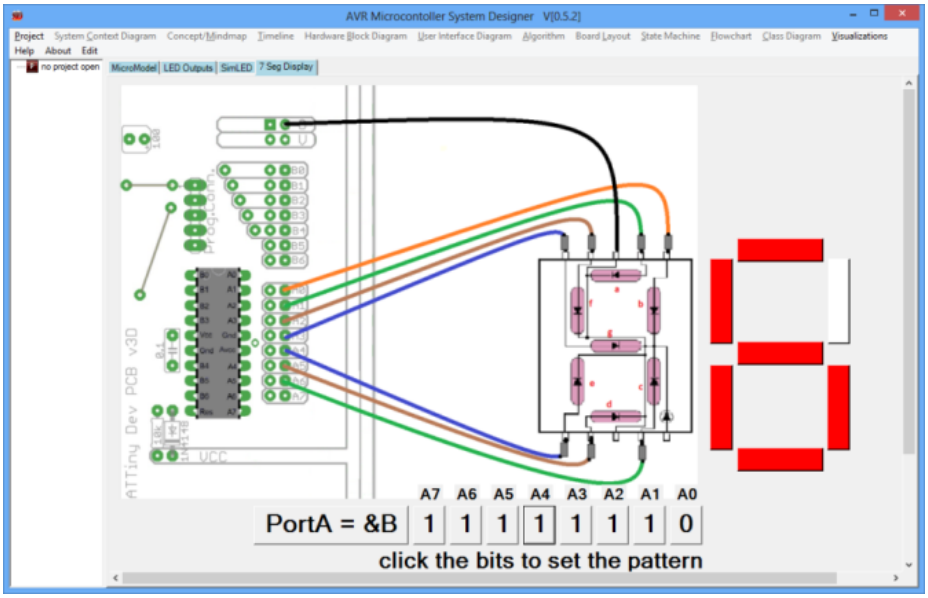


Figure 4.2: Visualisation of seven segment display coding patterns

The PWM visualisation, Figure 4.3, was developed to support a student who wanted to control the speed of a DC motor. I had explained the processes within a timer using a pen and paper explanation with limited detail from the datasheet, but the student was still confused. I developed the simulation and the student used it once and never referred to it again, though went on to make extensive changes to the timer settings using the datasheet and an oscilloscope to monitor the waveforms. Further still he explored other internal hardware registers of the microcontroller without any teacher support referring only to the datasheet. This is an interesting result as the visualiser not only provided direct support for the timer but also allowed the student to abstract learning about register control within a microcontroller and transfer it into novel situations.

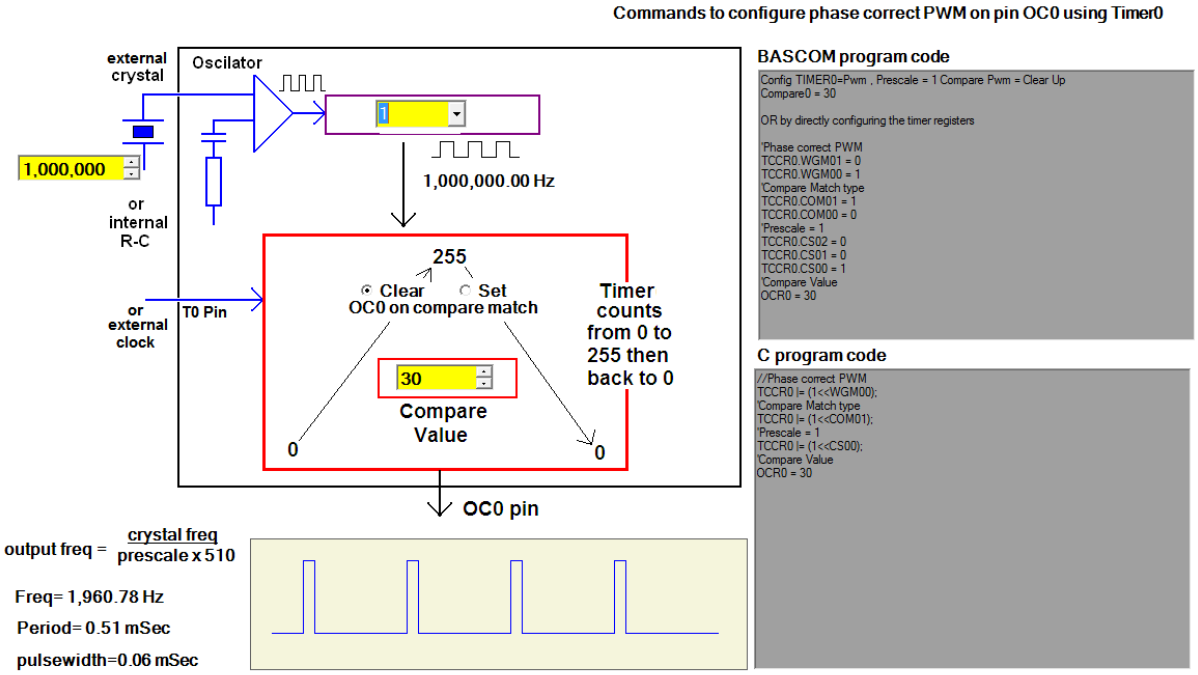


Figure 4.3: Conceptual model used to show PWM from an internal AVR hardware timer

A voltage divider visualisation was trialled with students to assist in learning calculations using Ohm's law. The initial diagram, Figure 4.4, shows no calculations; these are revealed in a seven stage process by clicking on the button on the right hand side of the diagram; the final state is shown in Figure 4.5. This tool was initially developed as a whole model which showed all the calculations at once. However students struggled with the amount of information in it and it was redeveloped into the part-whole sequence to reflect understandings of cognitive load theory (Chandler & Mayer, 2001). The result was that fewer students needed support in class with their own work after having used the part-whole model.

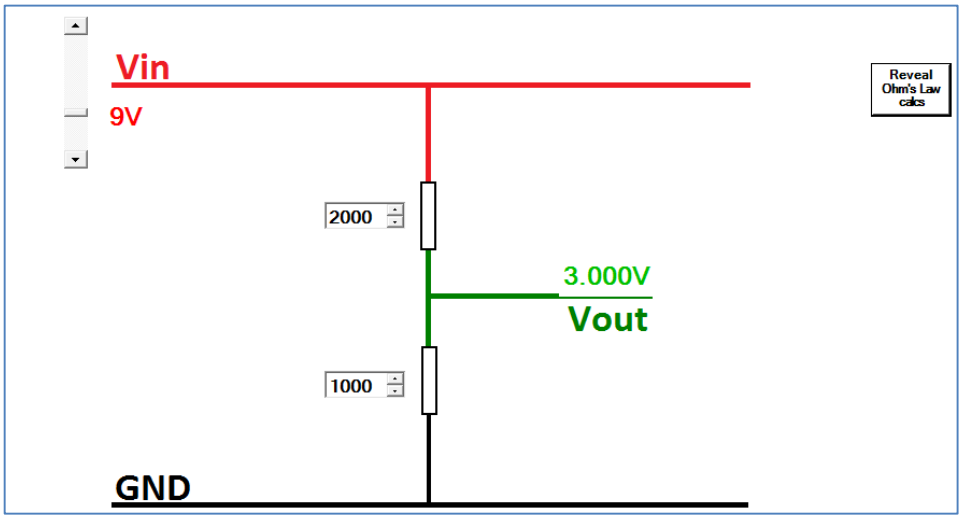


Figure 4.4: Voltage divider visualisation showing no calculations

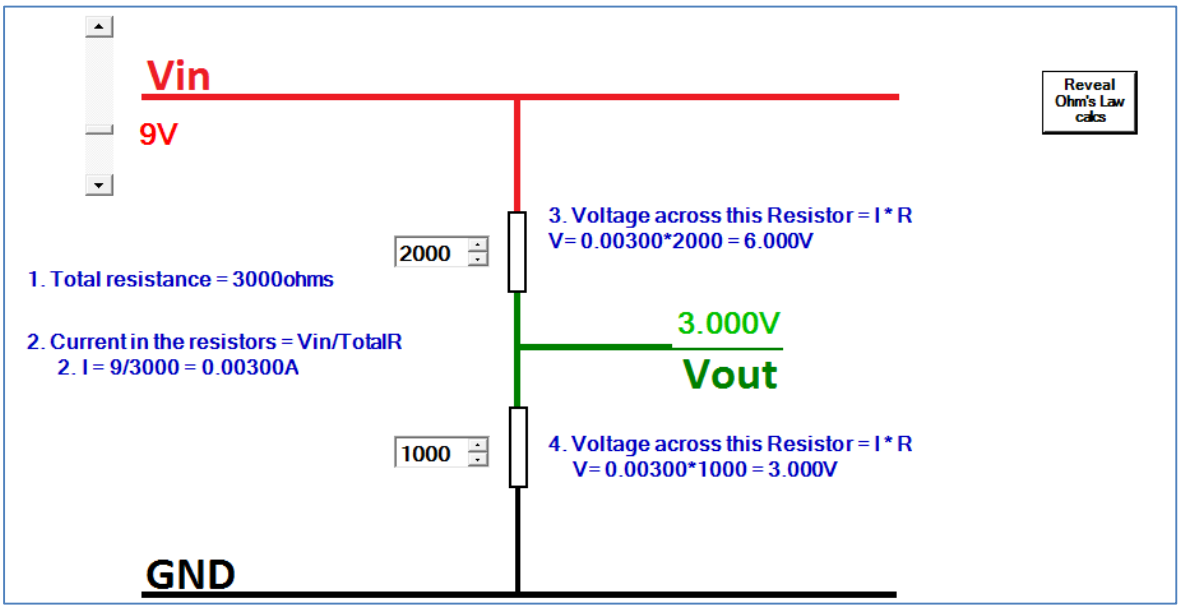


Figure 4.5: Voltage divider visualiser showing full calculations

The results of using visualisations in class are supported by more powerful studies into their benefits in learning to program (Lahtinen, Ahoniemi, & Salo, 2007; Lahtinen, Järvinen, & Melakoski-Vistbacka, 2007; Ma, Ferguson, Roper, Ross, & Wood, 2008), and agree with this research literature in several dimensions. Firstly, once a student has used the visualisation, for the most part they appear to have understood the concept and they progress with their own project satisfactorily without having to refer to the model again. Secondly, many students gain sound conceptual understandings and are able to abstract meaning (e.g. register use) and transfer this into other unfamiliar contexts. Thirdly, that the model must be close enough to the real situation so that the student can make links from the known to the unknown. Lastly, that simple models or models that progressively reveal information are easier to learn from as they control the amount of extraneous information and cognitive load.

An important issue for classroom practice, also relating to extraneous information, has been noted in class and powerfully supports the use of visualisation with students. It is rare during whole class instruction that all students will be able to work with their own hardware to undertake an exercise, as at least one student will have a microcontroller, interface, compilation or programming problem stopping them from focusing on the task. When using visualisations these issues do not exist thus allowing every student to be focussed on the learning associated with the visualiser.

Data from using these visualisations in class could be studied in detail to identify further aspects of visualisation, however many of the principles of visualisation have already been explored in research literature. The most powerful outcomes of this action research were the realisation that student learning benefits significantly from visualisation and that emulators and simulators that already exist were not satisfactory supports for novice learners of the AVR microcontroller based systems we use. The simple alternative would be to move away from the AVR to the popular school based system, the PICAXE. This, however, because of its technical constraints and costs, was not seen as a viable option. This consequently triggered an interest in a larger scale visualiser to support my concerns about programming understanding in the context of the AVR system and provided direction for this research project.

Chapter 5. Extending System Designer with the visualiser

This chapter begins with a brief summary of the pre-study state of System Designer's contribution to learning about embedded systems and programming understanding (section 5.1). The investigation of existing visualisation environments, their analysis against educational literature and the subsequent abstraction of their salient features to identify the best possible feature set for the visualiser is in section 5.2. These features are catalogued and summarised in section 5.3. The decisions made to enhance the visualiser extension are discussed in section 5.4.

A number of appendices were written to capture important aspects of the development of the visualiser, specifically its class structure, interrelationships and signal processing. This documentation was invaluable in planning, developing and maintaining the application, something which is stressed to students at school. Appendix H describes the pre-research state of System Designer's diagram tools, these are: mind-map, timeline, system context diagram, block diagram, flowchart, state machine, class diagram and sequence diagrams. A code map for the classes relating to the visualiser extension is in Appendix I. This appendix details the most complex class sequence interactions in System Designer, i.e. how the visualizer processes and visualises microcontroller interrupts. Appendix J describes the lexer and appendix K the recursive descent parser for the C language, while appendix L includes EBNF diagrams developed while planning the recursive descent parser. The BASIC parser is very similar to the C parser and is not described. Appendix M is a description of the run/pause/stop processing of the visualiser, a complex process involving cross thread communication between the GUI and the parser which is running in a background thread.

5.1. The existing learning capabilities of System Designer

It was clear from using System Designer with students in the classroom and from the increase in students' results that System Designer's existing features were successfully contributing to various aspects of student understanding; particularly in contextual and systems understandings as well as model based design and planning of their projects. Classroom action research on visualisation also provided indication of benefits for student understandings on some aspects of embedded systems learning and programming understanding.

Table 5-1 presents the features of the pre-research state of System Designer and how these supported embedded systems learning and programming understandings.

System Designer feature	Embedded systems learning and programming thinking
Standalone visualisations (LED, 7-segment, FET, ADC)	Minor aspects of hardware-software interaction and reactivity. Some support for the action level of programing thinking.
System context diagram tool	Contextual understanding of embedded systems.
Block diagram tool	Understanding of how an embedded system is constructed in terms of inputs and outputs.
State machine diagram tool with auto code generation	Some minor aspects of reactivity, limited because of its static nature. High level of support with the textual level of programming understanding. Strong benefits for student understandings and planning of larger programs.
Flowchart diagram tool	This had been developed and used in class with students in an effort to help structure program code; its inability to support the reactive nature of embedded systems had seen it put aside in preference for the state machine diagram tool. The literature has revealed its potential in informing student understandings at the important intermediate level of schemas, strategies and relational understandings that students need to develop (sections 2.3.4 and 2.3.6).

Table 5-1: Existing System Designer feature benefits prior to this research

5.2. Analysis criteria for existing visualisers

There are a great many visualisation tools available that can be used when helping learners, each expressing the creator's unique style, preferences and situation. While common features for the new visualisation tool could be drawn from others tools and prior experience with simulators a pedagogical stance to idea generation for the visualiser was seen as most strategic. Analysis of existing tools thus began with the review of visualisers as model-based learning environments rather than as a set of visualisation features. Eight dissimilar visualisation learning environments were selected as they presented as wide a range of learning approaches as possible. This, coupled with the literature review, led to a deeper appreciation of visualisers as learning tools and supported the abstraction of a feature set with the most potential to support student understandings.

While it would be useful to have a set of recognised measurements by which novice learning environments can be assessed, these unfortunately do not exist (Gemino & Wand, 2004; Guzdial, 2004). A range of understandings however do exist with regard to learning that takes place when using conceptual models. Much of this relates to cognitive processing theory and managing extraneous load and incidental processing (Mayer & Moreno, 2003). A range of characteristics for multimedia learning has been developed to analyse conceptual models, and these are presented in Table 5-2.

Complete	All needed elements, actions and associations for learning a concept are present.
Concise	A summary only capped at five or so common tasks to avoid excess information
Coherent	Sense making, showing interactions and rules for those interactions
Concrete	Depiction should be familiar to the learner
Conceptual	Shows the system operation meaningfully
Considerate	Student level vocabulary and organisation
Correct	Major analogies used are correct

Table 5-2: Summary of Mayer's (1989) characteristics of good models.

In analysing visualisers against the characteristics of Table 5-2 the key points that were addressed have been contextualised to address novice students' needs in programming understanding and embedded systems learning.

Completeness relates to how accurately the tool portrays the crucial embedded systems understandings such as interrelationship between hardware and software.

Conciseness is to avoid cognitive load issues, and follows the principal of reducing incidental processing (Mayer & Moreno, 2003) by removing extraneous information and keeping only that which is essential (Kester et al., 2010).

Coherency assists learners to make a sound model or representation of the action of an embedded system; this requires them to hold a number of things in working memory at one time. Having to move backwards and forwards between multiple windows or views of a representation breaks coherency. This is a key issue that instigated this research; novice learners struggle to understand as they have to move backwards and forwards between disparate hardware and software when using development systems.

Concreteness is when a learner can see aspects of a model that they are already familiar with and then they are able to start forming links to existing knowledge. Familiar items and links do not count against the number of new items that need to be processed by a learner when using a model but instead are positive as they form relational anchors for the new concepts.

Conceptual is when students are able to conceptualise understandings or make meaningful connections correctly, then they are able to actuate them when needed and transfer them into novel situations.

Consideration relates both to the tool and the use of the tool and has significant bearing upon reducing cognitive load, which is a key aspect of improving students' performance (McCartney, Boustedt, Eckerdal, Sanders, & Zander, 2013). The tool must be congruent with the level at which the student is operating; a novice will struggle with too many new words, images and interactions, compared to a more experienced learner (who can cope with both more and fuller detail). The first goal of this research is to be congruent with the needs of the novice over the more experienced learner but ideally also to provide steps for more significant understandings later on.

Correctness is how true the tool is to the actual. When a tool represents a complex object to a novice there will be a necessary condensing of the object. False understanding must never be conveyed as misconceptions are more resistant to change than development of new understandings (Goris & Dyrenfurth, 2010). Teachers that use conceptual models must fully

comprehend the limits to the analogies they present and only communicate the appropriate understanding (Coll et al., 2005; Gilbert, 2004).

5.2.1. BASCOM-AVR

BASCOM-AVR (MCS Electronics, 2013) is the integrated development environment (IDE) currently in use in the learning programme. It has a built in simulator as shown in Figure 5.1, which models many aspects of the inputs and outputs as well as memory. It also has a limited hardware debugger where a development board can be connected via serial communications to the IDE and the results of the software directly observed on the hardware. The key feature of simulators is that program code can be stepped through line by line or run at speed. This feature allows users to analyse issues between hardware and software; for learners, this ability is a key aspect for enhancing meaning (conceptual benefits), and sense making (coherency) of program code. The operation of program code at a slower rate does have an impact on correctness, as the consequences of programs running at full speed are not always observable at a slow speed.

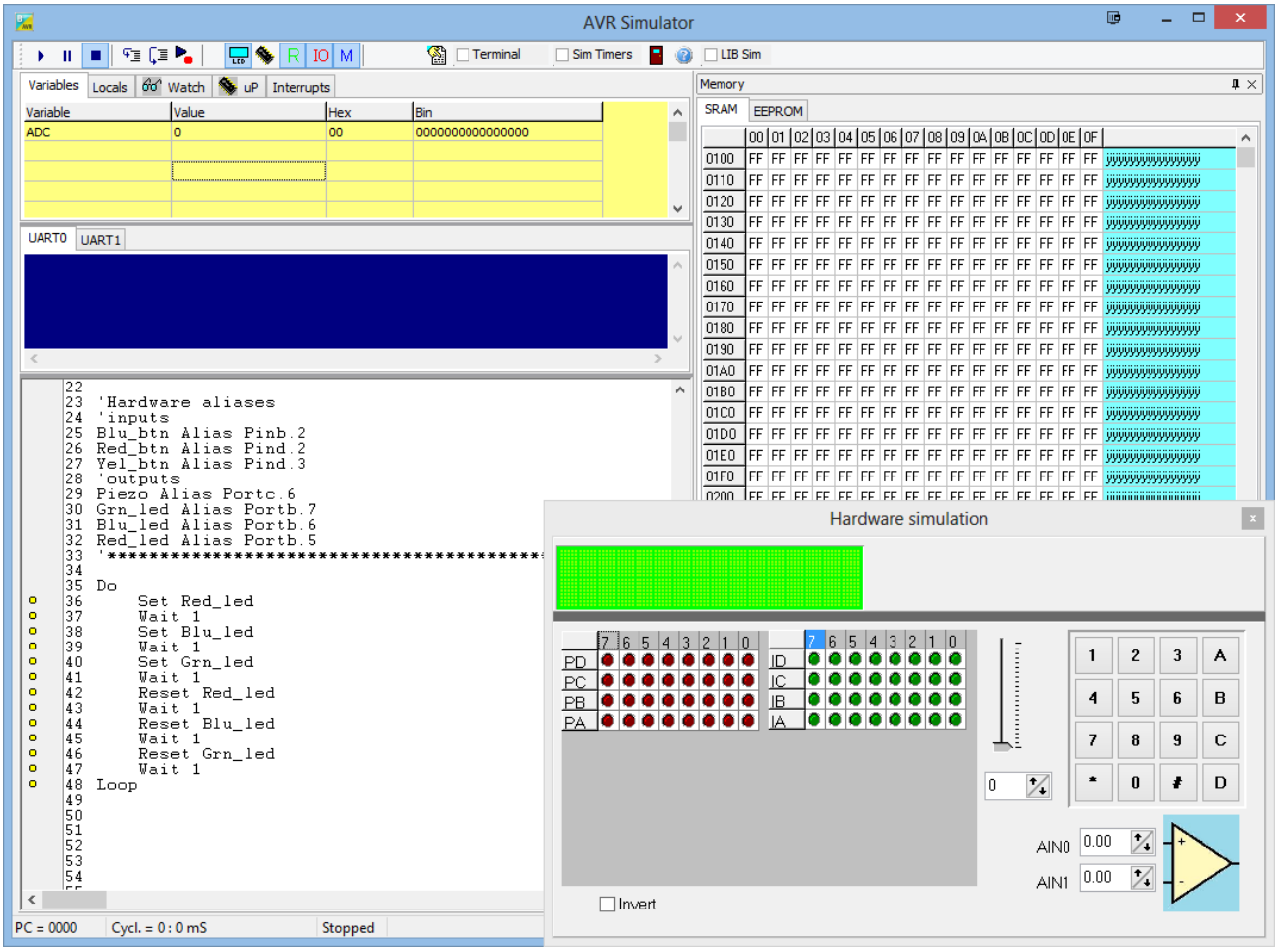


Figure 5.1 BASCOM-AVR simulator

Comments collected while novice students used this in class related to it as being too complex (conciseness characteristic) with many unfamiliar abstractions (coherency issues); they were unable to relate it to their own hardware (concreteness characteristic); it had too many unfamiliar terms (consideration characteristic) and students ultimately showed limited understanding of their own systems (conceptual characteristic). One aspect of confusion experienced by students when simulating larger programs was that program flow can be difficult to follow when it changes between several function calls.

With regard to the key concepts of real-time embedded systems of concurrency and reactivity, it presents some support of reactivity, with users being able to set a slider for input to the ADC and trigger inputs to mimic environmental changes. Learning about variables and memory are handled through two windows, one presenting a grid view of SRAM and the other presenting a view of selectable variable values (yellow area in Figure 5.1). Having both views is highly considerate of the novice learner; however students did need specific instruction to interpret the views.

5.2.2. Atmel Studio 6

Atmel Studio (Atmel Corporation, 2013a) is another full IDE with integrated simulator as shown in Figure 5.2. It also incorporates debugging of hardware using a range of communication protocols between the IDE and the microcontroller development hardware. Atmel Studio simulates C and assembly language programs.

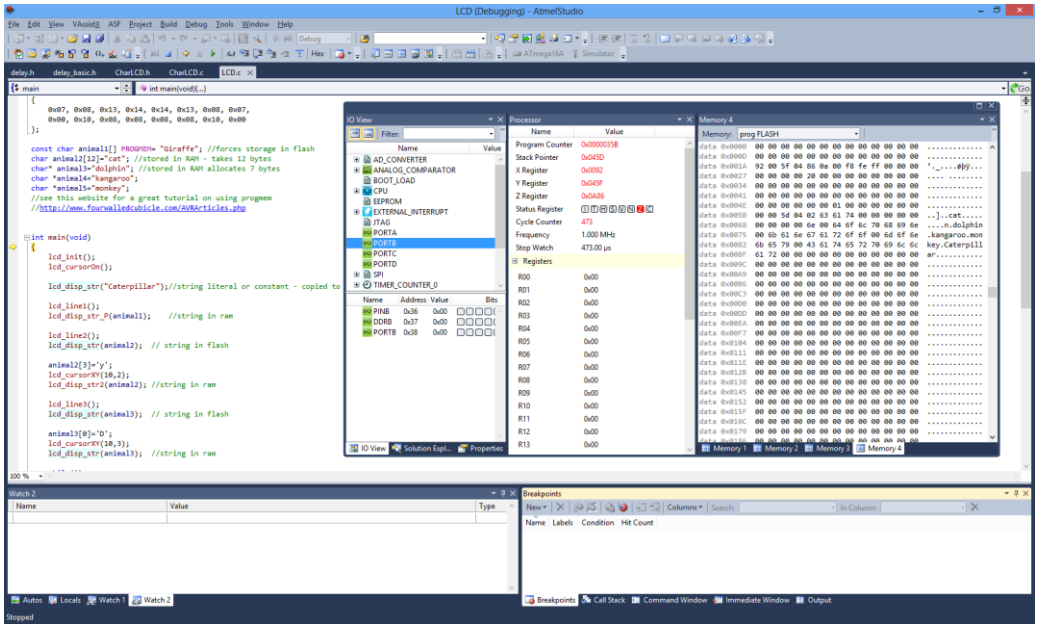


Figure 5.2 Atmel Studio 6 simulator

This software is comprehensive and fully aimed at professional use. On four occasions it has been used with students in year 13. Each was a highly capable student who had familiarity with Visual Studio (which Atmel Studio 6 is built upon) and some experience with programming in C or C#. These students all found little difficulty becoming familiar with it.

For a novice its completeness and general correctness as a professional tool means that it lacks conciseness and creates coherency difficulties for new learners. The high amount of unfamiliar information in professional tools such as this is a significant issue of concreteness as it overwhelms a novice learner. The ability to extract the important detail from such a complex tool makes gaining conceptual understanding difficult. With regard to correctness the online documentation (Atmel Corporation, 2013b) describes some current limitations of the simulator: non-volatile fuses and EEPROM are not simulated; it has no analog periphery; and not all AVR devices are represented. The simulator can be extended with stimuli files (text like scripts) which can be written to model some aspects of real world scenarios supporting aspects of correctness,

completeness and coherency. As well as the ability to highlight the current line of code, the simulator has a trace output which records the sequence that the running program has taken, which supports the development of conceptual understandings of program flow. Its ability to interface directly to hardware as a debugger has not been trialled with students but will support conceptual meaning for experienced students.

5.2.3. PICAXE

The PICAXE suite of tools is highly developed and is primarily targeted at the novice and school market. The PICAXE Virtual System Modelling (VSM) circuit simulator (Figure 5.3) is one of the tools available; it is a fully capable tool and incorporates Berkeley SPICE analysis and virtual instruments (Revolution Education Ltd, 2013).

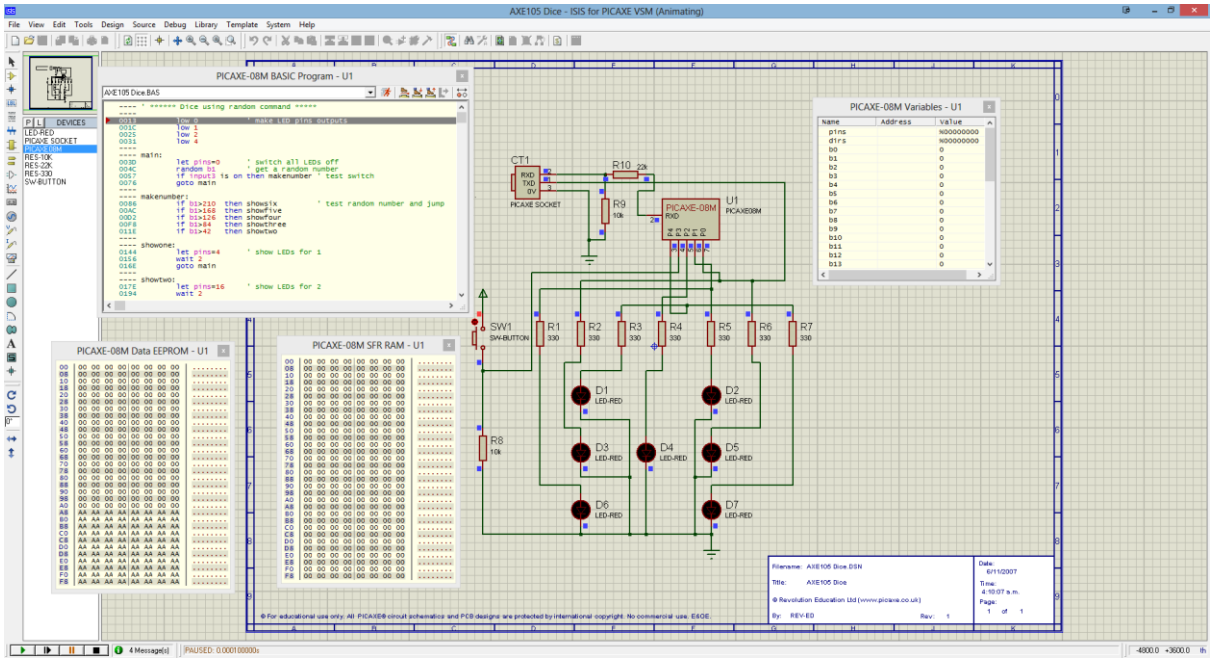


Figure 5.3 PICAXE VSM Circuit Simulator

The tool, whilst being complete, can still be easily configured as a concise and coherent learning environment for novice learners. Its diagrams and simulator directly model a full circuit making it concrete and good for developing conceptual understandings. Because of its focus on learning, the limitations of the PICAXE system as a first learning tool are few. The prominent limitation with it is related not to the learning environment but to the commercial decisions made with the PICAXE development. The PICAXE is a standard PIC microcontroller with a proprietary bootloader programmed into it by Revolution Education prior to sale. To take advantage of the VSM and other PICAXE applications users have been limited to the subset of features that the designers have chosen and to the use of the BASIC language via their interpreter.

5.2.4. PICsim

PICsim (Lopes, 2013) emulates six different development boards for PIC microcontrollers from a range of three different suppliers; one of the boards is shown in Figure 5.4. Each board has pre-set input and output devices and no configuration is available to users. Having a direct visual correlation with known hardware makes the emulator highly concrete and enhances its correctness as it shows running code accurately in the context of that board.

The program code is loaded as pre-compiled HEX files so the source code and board emulation is not presented as a contiguous flow. This gap between software and hardware loses conciseness as the consequences of individual lines of program code cannot be viewed. This means it is not easy to make the connections that novice learners need (completeness) thus reducing its benefit for conceptual understanding.

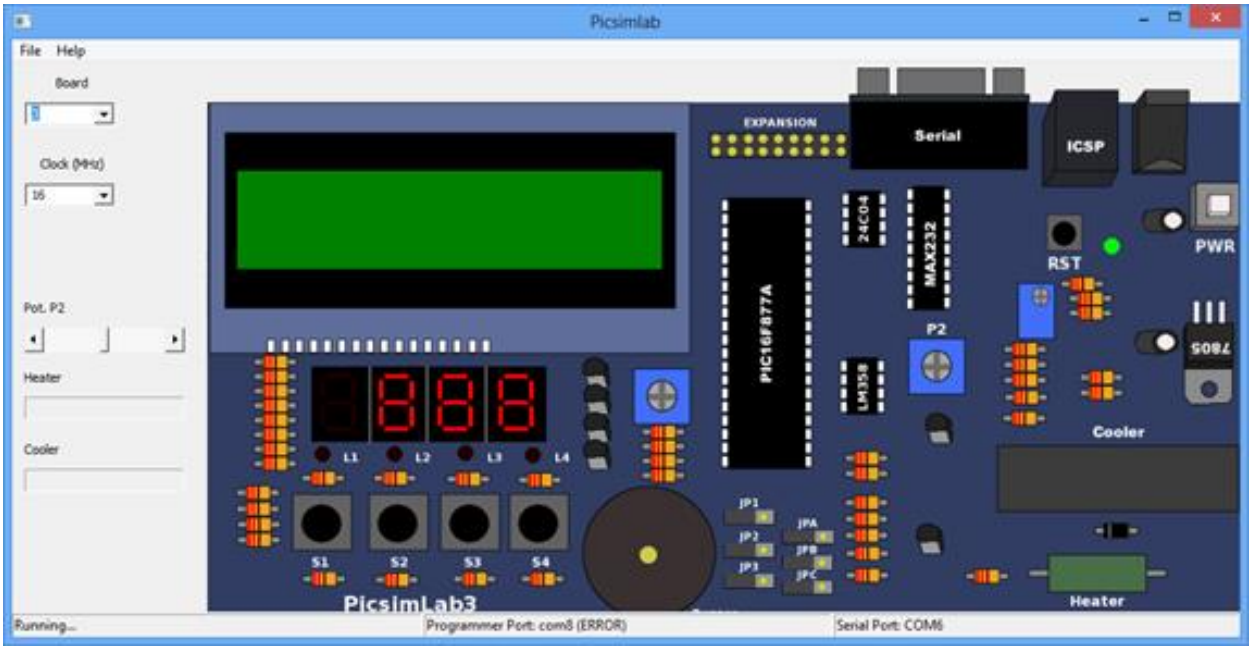


Figure 5.4: Picsimlab

5.2.5. ViLLE

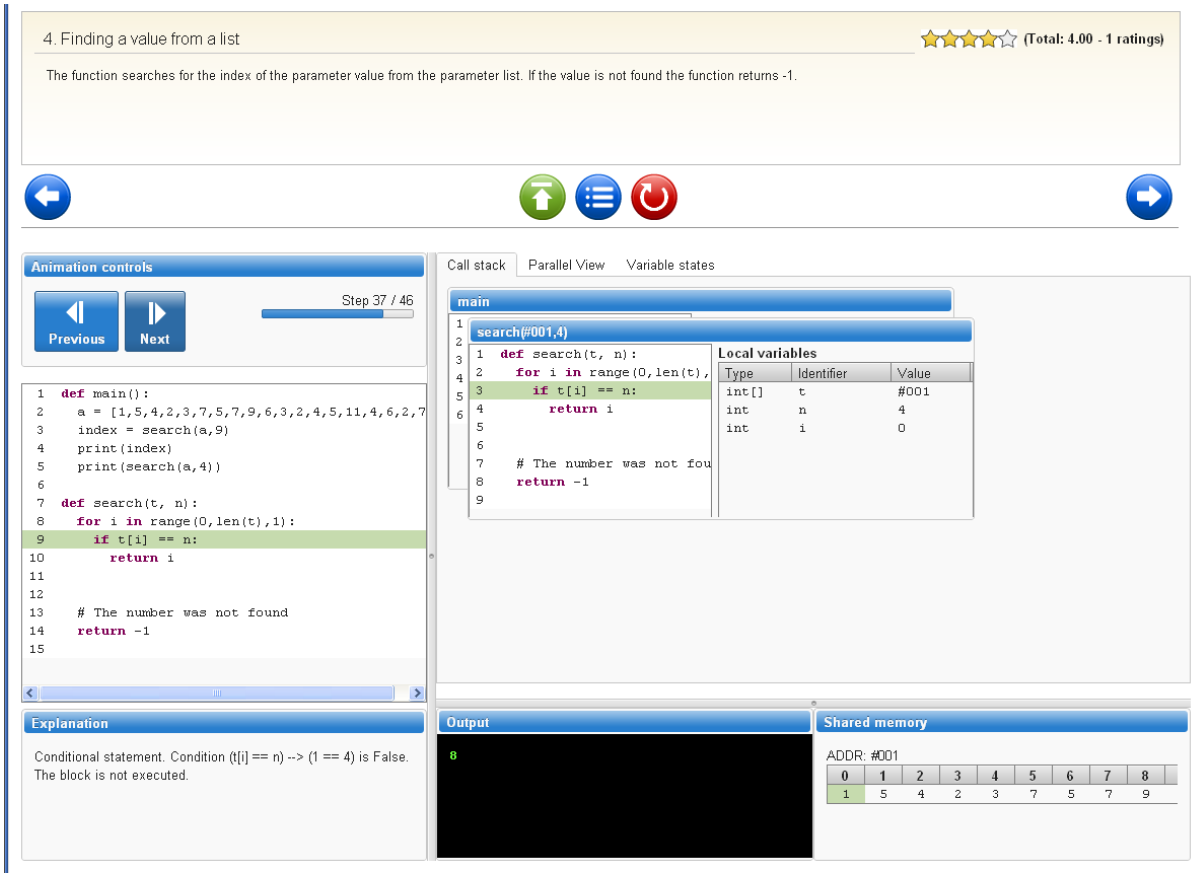


Figure 5.5 ViLLE collaborative learning tool

ViLLE (Laakso, 2013) is a learning specific tool for learning to program and is not aimed at embedded systems (Figure 5.5). It is relevant to this research as it has a number of learner specific additions that have been shown to be effective (Laakso, Kaila, & Salakoski, 2008). One aspect is that ViLLE is much more than a simulator; it allows educators to create courses and assessments as well as collect data about student learning. The simulation can auto-generate line by line explanations and allows the inclusion of pop up questions to support students' sense making (coherence and consideration). ViLLE decomplicates the visual memory model avoiding the excess information found in hardware simulators thus improving conciseness. It has a parallel language view aimed at allowing learners to generalise learning more quickly than when using a single language enhancing its coherence and conceptual characteristics. The call stack can be viewed as different windows showing the different scopes that code is in, thus also enhancing coherence and conceptual characteristics. The ability to both step forwards and backwards through running code could be seen as both breaking the characteristic of correctness, as programs do not run backwards in time, and increasing the conceptual learning characteristic as code execution can be reviewed quickly without the delay or having to run the program again.

5.2.6. Raptor

Raptor (Carlisle, 2013) is a flowchart algorithm visualisation tool; it visibly executes the flowchart (Figure 5.6) making it useful for sense making (coherency) and it has a simplified memory model (conciseness).

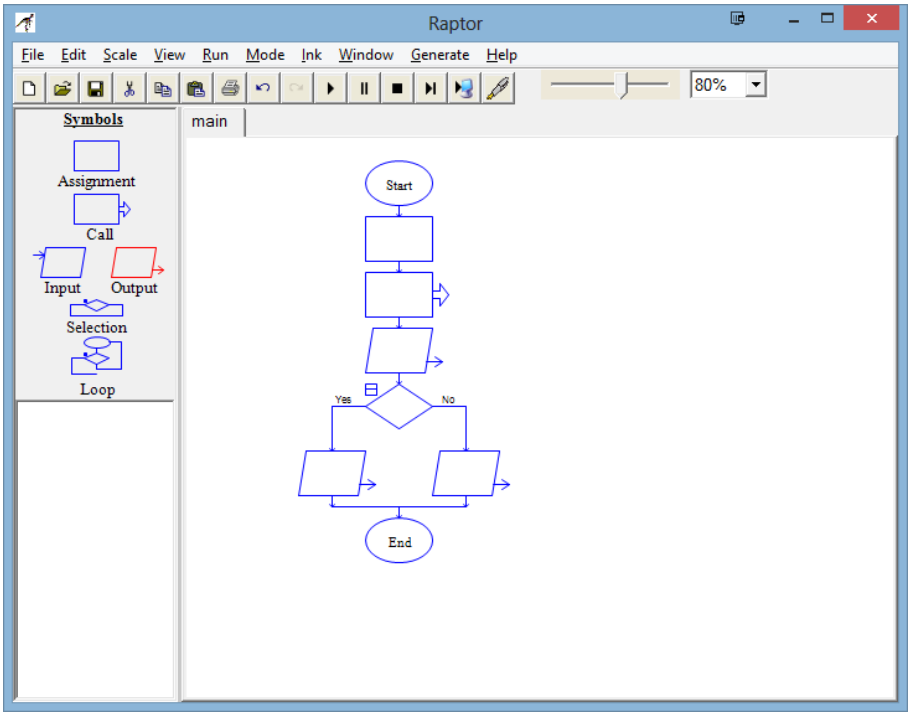


Figure 5.6: RAPTOR flowchart visualiser

One of its main strengths is it is highly considerate on novice learners with its low number of visible details. It has a progression of complexity through several modes: a global address space, local variables and parameter passing and object oriented modes. The goal is to abstract algorithm detail from syntax. This affects its correctness as it has no code editor and is not language centred. It does incorporate a wide range of software concepts including arrays and procedures adding to its completeness. The value of flowcharts as a tool beyond algorithmic explanation is limited in the environment of real time and reactive systems (Harel, 2009) so this tool is seen as having lower conceptual characteristics in the context of this research. Visual programming environments are also limited in their ability to support learners' real (textual) programming contexts (Parsons & Haden, 2007) thus having a low coherency characteristic.

5.2.7. ProgrAnimate

ProgrAnimate (Snow, 2013) shares several characteristics with Raptor in that it's a flowchart visualisation tool (Figure 5.7) and developed in a desire to overcome algorithmic problem solving difficulties (Scott, Watkins, & McPhee, 2008). It achieves this by linking the graphical and textual models of programs. It has a more complex model of memory (with types) and supports multiple languages as well as syntactically simplified versions of Java and Visual Basic. It is included here as it allows students to alter the run speed of the animation (making it coherent), the view of variables includes types (completeness) and has the ability to both step code forwards and backwards as ViLLE does.

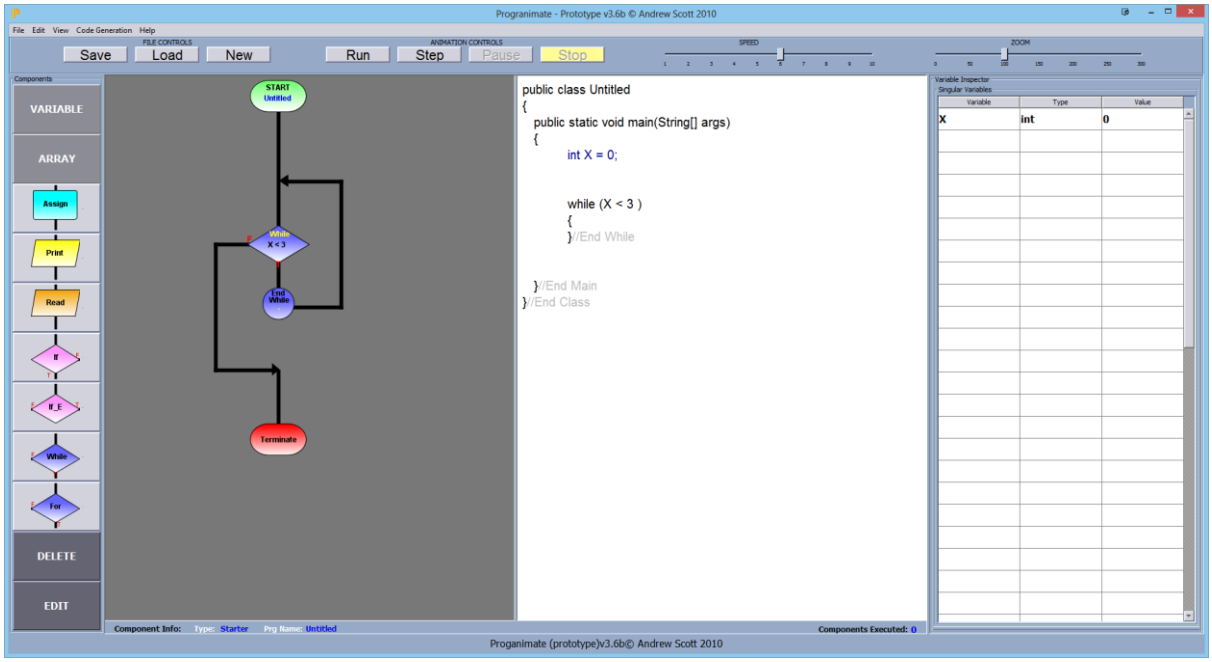


Figure 5.7: ProgrAnimate visualisation tool

5.2.8. eBLOCKS

eBlocks (University of California, 2013) is a block based hardware development system which includes a software simulator (Figure 5.8). It is not focussed on the novice learner so much as on "ordinary people", those with no programming or electronics background or interest in pursuing one (Cotterell, Vahid, Najjar, & Hsieh, 2003). As such it represents an exciting opportunity to make the mystery of embedded systems more accessible. The simulator environment has many strengths. It shows a very familiar representation of the physical environment (making it highly concrete) with colours and animations revealing the operation of a system (coherence); its electronics and program details are abstracted away into blocks making it very concise. The environment is highly novice-centric with little extraneous detail (considerate). Due to its high level of abstraction of the detail, a novice learner (in contrast to a novice user) would find it reduced in correctness and completeness and not able to support development of the understanding they need.

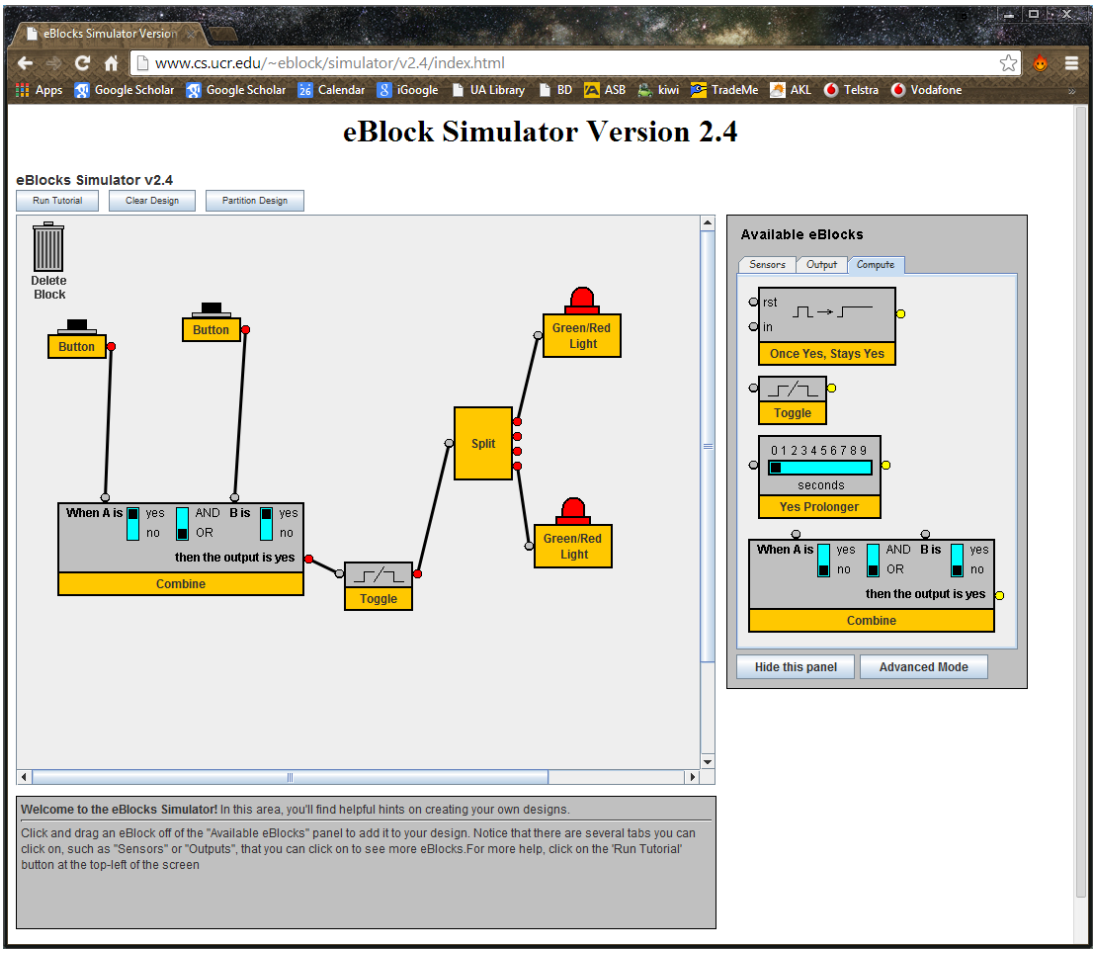


Figure 5.8: eBlocks simulator

5.3. Summary of features

Whilst all the tools investigated have many features, some of these are highly important for novice learners and some of them are actually counterproductive, actively working against novice understandings. Figure 5.9 lists the identified features of visualisers and graphically shows how each measures against the characteristics for good models.

Each colour represents one of the seven different characteristics of good models. Every feature is scored against each characteristic; where the feature was identified as positively influencing a characteristic it is on the right side of the graph, if it works against a characteristic it is on the left side of the graph. The centre of the graph is neutral or no effect.

It can be seen that while many features are advantageous for novice embedded systems learners, some are potentially harmful in terms of model based learning. The order of the features in the graph represents the importance of the features for incorporation into this visualiser. Of all the features investigated the last five items on the list were not implemented; two of these because of their perceived negative impact and while the other three could be shown to have benefits, they do not represent the core purpose of this research.

Some features were seen as having many characteristics and some fewer; only the feature of hardware animation, which is at the core of the visualiser, was seen as influencing all seven characteristics. This is at the top and at the bottom of the graph. The crucial point of difference is whether the hardware animation is contextually close to the software animation. At the bottom of the graph there is no contextual relationship between the program being run and the hardware. Countering this disconnected effect is the underpinning reason for development of this visualiser. This disconnected mode of visualisation is exemplified by the Picsimlab visualiser (section 5.2.4). This tool should not necessarily be seen as negative as it was developed with a specific goal in mind. Its inclusion however is purposeful as it serves to draw attention to the importance of careful, almost pedantic, pedagogical discourse when analysing the fitness for purpose of any educational tool.

For those features selected for the visualiser only the two characteristics of correctness and conciseness are found on the left of the table. Four of the features are essentially incorrect, for example single step and slow run animation of program code are incorrect as program code does not run line by line waiting for user interaction (or at extremely slow speed). These highlight the crucial realisation that models are analogies and representations, and educators must explicitly

communicate their limitations with learners so that the learners do not gain misconceptions by learning the model and not the real system (Coll et al., 2005)

The second disadvantage that presents often in the graph is conciseness; for the 19 features selected for the visualiser it was seen as having nine negative and four positive effects. Some features were seen as not being concise because they present a great deal of information for learners, for example scope, trace and timing displays. To mediate issues with conciseness, incidental processing and cognitive load, these features were included in the visualiser but as options rather than as normally visible. Contextualised hardware animation, at the top of the list could be seen as working against conciseness, as it is aimed at the highly complex understanding of the interrelated nature of embedded systems hardware and software. For this reason the animation of hardware was decomplicated as much as possible in the development of the visualiser by leaving out fuller circuit analysis characteristics (as found in the VSM Picaxe simulator) and introducing a simplified use of colour to highlight high or low voltage potential.

One aspect of the analysis that is not seen easily in the graph is that one characteristic (concreteness) does not feature highly, having only 7 effects in total. This draws attention to the low overall concreteness of embedded systems learning as it is a hidden technology and something people are unfamiliar with.

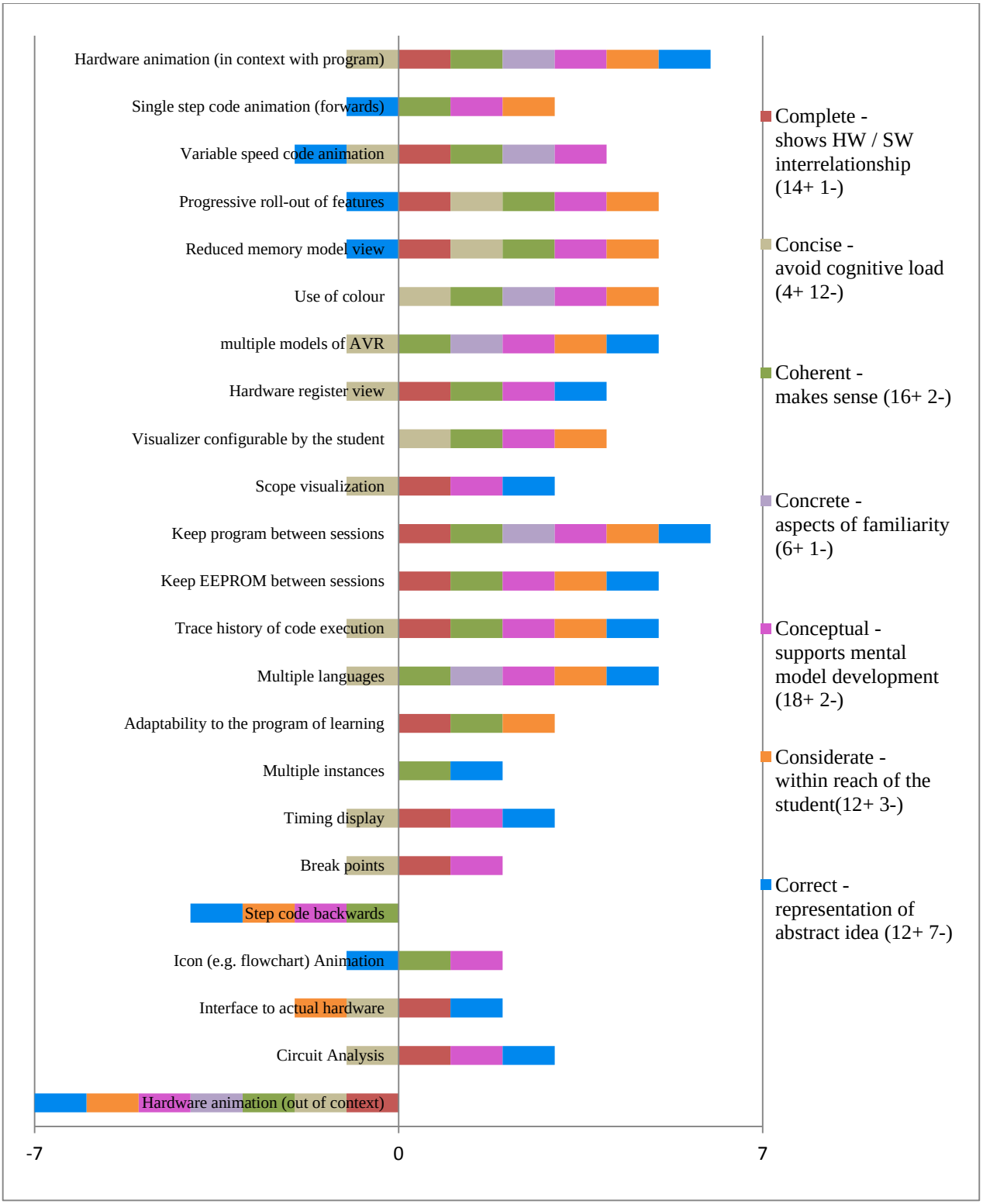


Figure 5.9: visualiser features measured against model-based learning characteristics

5.4. Visualiser design decisions

5.4.1. Pedagogical decisions

The purpose for developing System Designer has always been to provide integrated novice-friendly systems by reducing the amount of disparate and seemingly unrelated pieces of information that students need to cope with. For this reason the new visualiser tool was conceived as an integral part of the existing System Designer application rather than as a separate entity. As students' learning progresses, more advanced conceptual understandings are realised by progressively exposing more complex features. In this way students can start with the most simple of embedded systems (a switch and LED such as in Figure 5.10) and see how the hardware and software interrelate (Figure 5.11) without changing to another diagram and without extraneous information. Students can then progress to more complex systems (Figure 5.12) easily by adding further I/O devices. During the learning process extra features of the visualiser environment are initially hidden (Figure 5.11) or exposed as required (Figure 5.13).

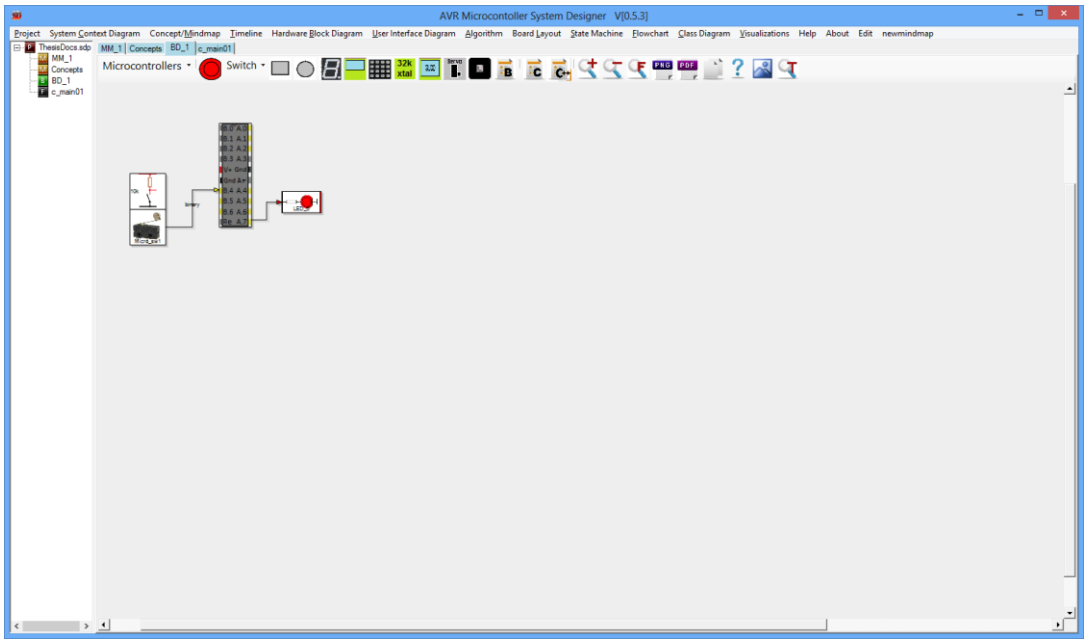


Figure 5.10: System Designer block diagram editor

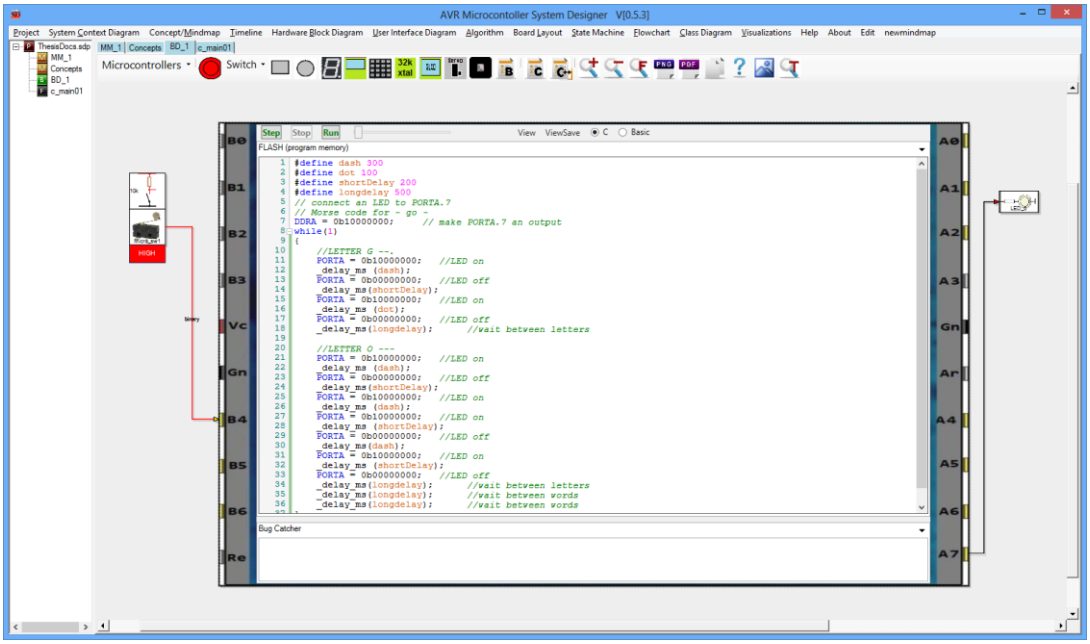


Figure 5.11: System Designer block diagram editor with visualiser open (simplest view)

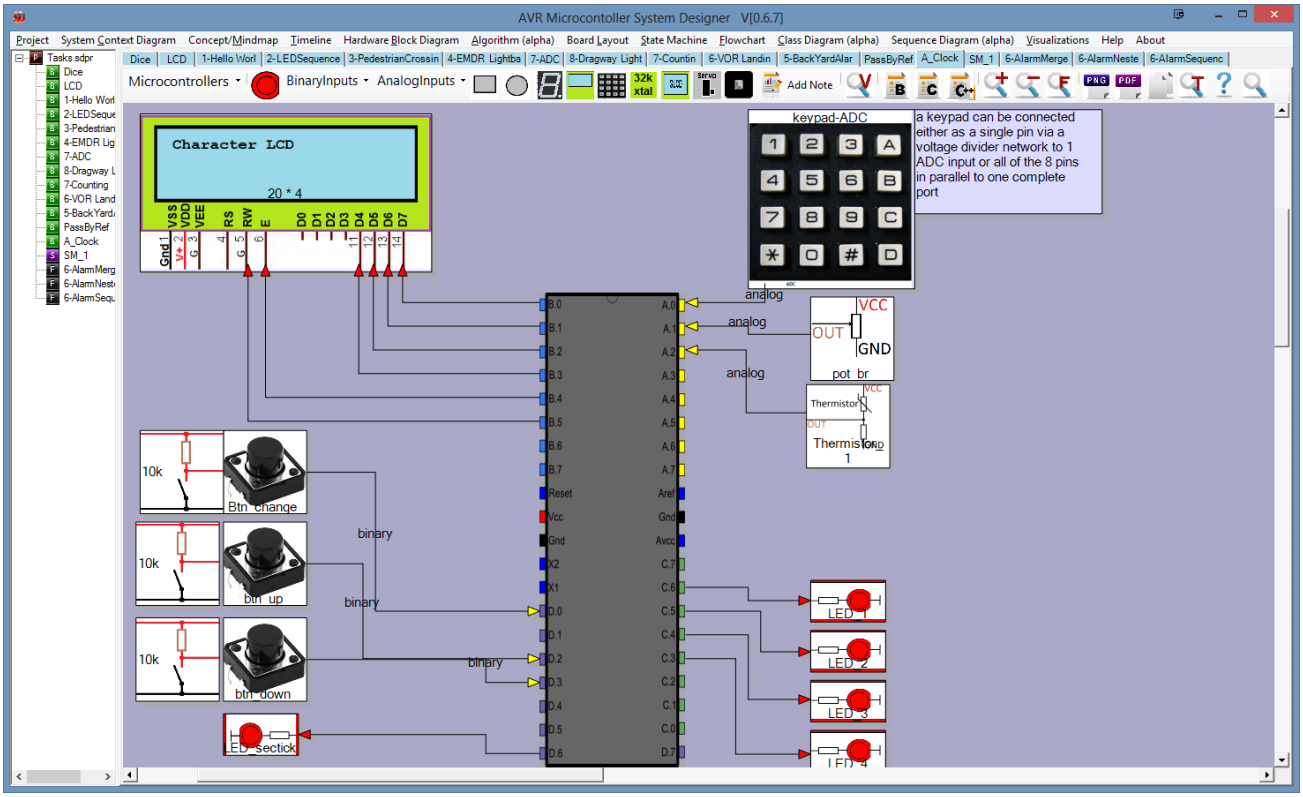


Figure 5.12: Complex embedded system

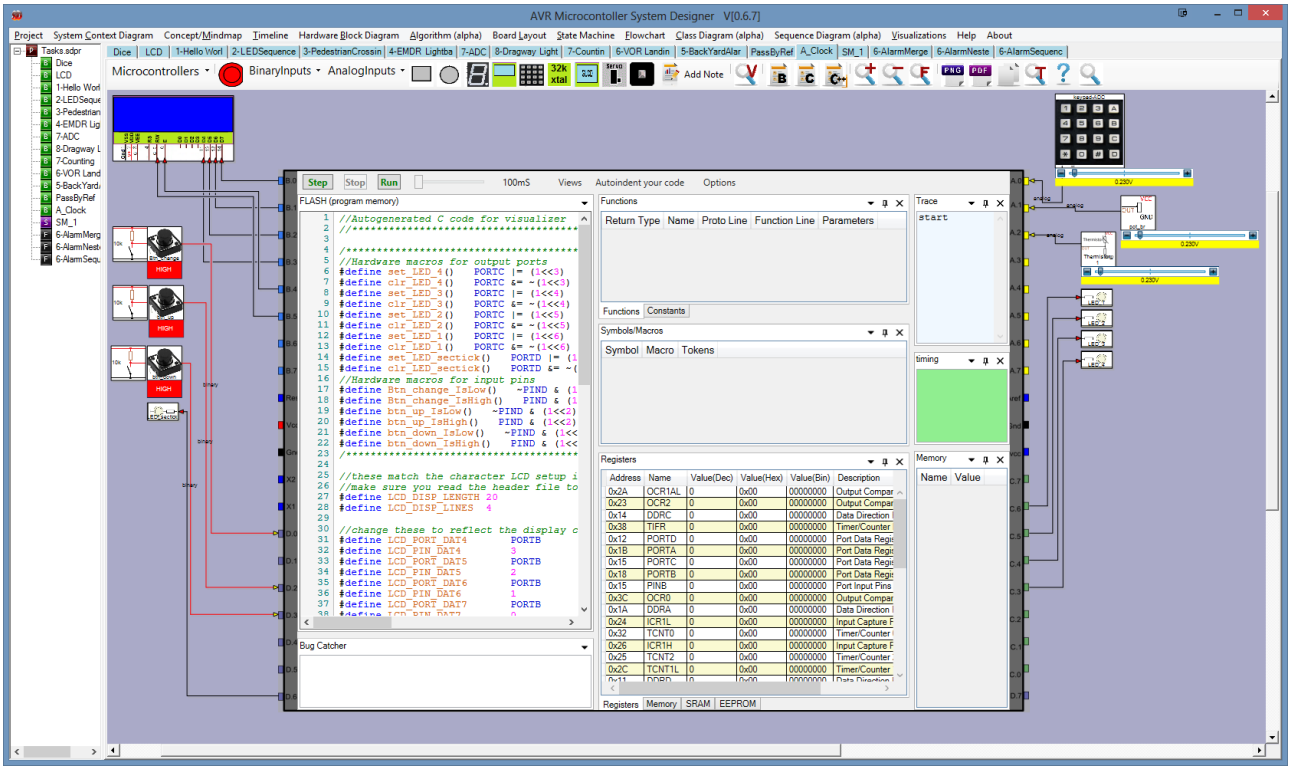


Figure 5.13: Complex embedded systems with all visualiser features shown

This is important in terms of learning being within reach of a student, which is a model of cognitive human development known as Vygotsky's zone of proximal development (Chaiklin, 2003). A proximal zone represents new meaning and understanding which is accessible for the learner; however it is only accessible if they already have the prerequisite understanding and suitable external assistance to help them bridge the gap. Any understanding which is too advanced will be outside their zone of proximal development and will need intermediate steps to keep the learning always within reach of students. This has been noted clearly in the classroom via action research into the success of using simple visualisations which progressively revealed information.

5.4.2. Technical decisions

The addition of the visualiser required a degree of review and restructure of some of the 30,000+ lines of program code which had been written prior to this research to create System Designer. These were extended by writing a further 15,000+ lines of code for the C and BASIC lexers and parsers, the visualiser classes and the various support classes. The visualiser extension required the integration of a number of code libraries including: Fast Colored TextBox (Torgashov, 2013) for the code editor window, observable concurrent dictionary classes for features of user program

code (RAM, functions, registers and constants) and Avalon Dock to manage the user interface (Avalon Dock, 2013). While System Designer was initially developed purely as a Windows Forms application, Avalon Dock is a library of Windows Presentation Foundation (WPF) classes. The observable concurrent dictionary classes were used as thread-safe collections because of their WPF binding properties.

The visualiser has been implemented with features that align with the analysis of features from other visualisation tools using the characteristics of good models.

The windows shown in the visualiser are fully configurable by the user (Figure 5.13). All windows are drag and drop. A number of views have been preconfigured and can be selected using the View menu. Most of these are user configurable and can be saved using the View sub menu.

Program code can be single stepped line by line with each line highlighted after its execution. Highlighting of code was chosen as after line execution rather than before so that learners would relate the highlighted line with the current state of the system. The program code can be run automatically with delays between each line of code as determined by a slider control setting. This is settable from 0 to 2000 milliseconds in 100 millisecond increments. When the slider is set to a delay of zero, visualisation of the outputs and tables representing memory and registers is slightly delayed due to update issues with the GUI related to cross-thread processing. At any time during visualisation the code can be swapped between step and run mode by simply pressing the desired button.

A break facility has been included; clicking on the column with the line numbers in or pressing ctrl-B will add a break point at a line of code where the cursor is (while clicking on the break point or ctrl-shift-B will remove it).

Currently the ability to step backwards through code has not been implemented. This has not been included as it was viewed as incorrect modelling for students particularly with regard to the aspect of embedded systems being real-time and environmentally reactive. Little research into this feature was found, however one study concluded that for algorithm visualisation it was not found to provide increased benefit (Saraiya, Shaffer, McCrickard, & North, 2004). The development of this feature may be considered in the future by integrating a database to store steps in the code (this may become a useful feature for logging of student work for analysis purposes).

Syntax colouring has been implemented, so that all lines with Boolean tests are coloured as to their result: red for false, green for true. The implementation of colouring is to direct the students' attention solidly onto the animation as students tend not to focus on the animation but the text (Nevalainen & Sajaniemi, 2006).

Hardware interfaces are dynamically visualised with the support of colour. An LED when reset (taken low) will have no background colour and when set to high will return to the colour it was defined with (e.g. compare the LED images in Figure 5.10 and Figure 5.11). The line linking the microcontroller and the LED will also change from black to red to show voltage potential. With switches the input link between the switch and the microcontroller will also change colour. In Figure 5.11 the link is red indicating a high. Attached to the switch image in the visualiser are two other items, a button and an image of a switch in a voltage divider. When the button is pressed the voltage divider image will change along with the colour of the link to the microcontroller. Three different types of switches are implemented to represent a range of switch types used by students in projects: push to make and release to break (push button and tact switch), push to make and push to break (toggle, micro and reed type switches), and push to make and automatic break (simulates a passive infrared detector-PIR).

A number of analog input devices commonly used in the classroom were developed (Figure 5.14). The graphics attached to these serve as indicators for the students as to how they are used, i.e. the LDR (light dependent resistor), thermistor and FSR (force sensitive resistor) are used as part of voltage divider circuits.

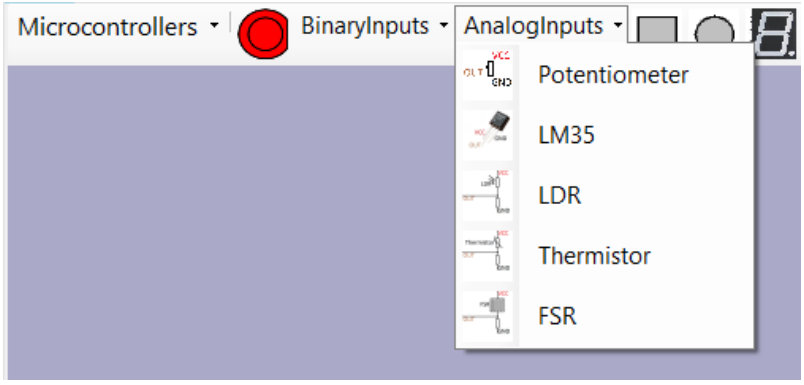


Figure 5.14: Analog Input menu

A timing trace is implemented and can be displayed live as the code is executed. The frequency of count is based upon the microcontroller clock setting which can be changed by the user. This is not an accurate count, as the code is interpreted within the visualiser and not compiled to machine code, but it represents a general indication of timing durations. Even though this is not an

accurate display it will be useful for supporting students' understanding of environmental timing constraints and concurrency.

Functions and parameter passing have been implemented. Function prototype definitions are required for functions that are not defined prior to use just as they are in C programming. In the memory model, names given to local variables reflect the scope they are within both as a number and as the name, so a variable 'x' declared with the main function will be named (1:main)x. This makes the concept of scope visible and supports learning such as recursive functions, something which novices find very difficult (Clancy, 2004).

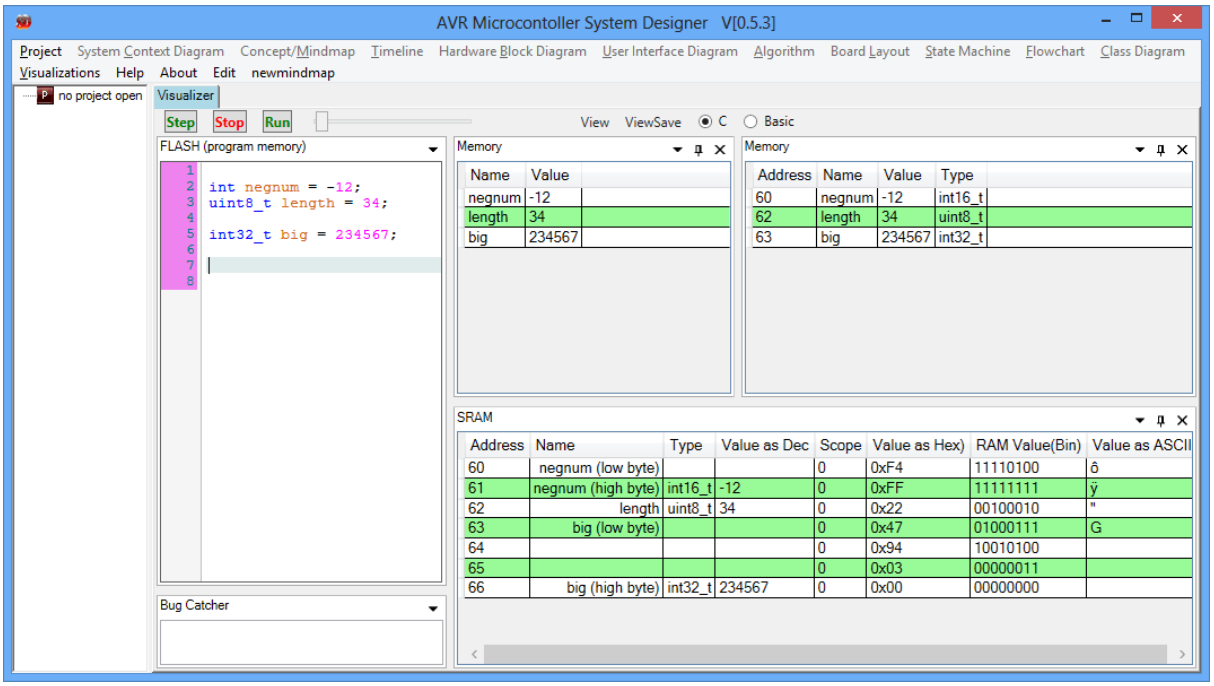


Figure 5.15: The Visualiser's three options for viewing memory

Memory models in microcontroller simulators are too complex for novices, and to support different stages of development in learning, three different views of memory have been developed (see Figure 5.15). The most simple has the name and value, the second has address, name, value and type, the third is a very full model where each byte in RAM that has been used is shown. All three memory views display scope. The most complex view has binary, hex, decimal and ASCII views of memory as are commonly found in simulators. A further feature of this is that a 32 bit type is split into its four bytes and each is shown as it would appear in a big-endian system. The AVR does not really have an endianness as most register are single bytes, endianness is a feature of the compiler and GCC is big-endian on AVRs. When a variable is declared in C it is given a random value, this is in order to support students in remembering to initialise it.

Two languages (C and BASIC) have been implemented in the visualiser; having two languages reflects that programming language choice does not affect ultimate programming ability (Chen, Monge, & Simon, 2006). It is hoped this will make the tool more useable for educators at both the secondary and tertiary level.

Various microcontroller registers are viewable in the visualiser, these include the port, pin and data direction registers as well as the registers for configuring the ADC, interrupts and timers. These are all set via in an XML file which is configurable by users.

Rather than one large symbol table for all functions, constants, macros and symbols, these have been implemented in three separate views. This is to support students separating the differences of each in their own mental model and reduce cognitive load during visualisation.

A program trace feature has been implemented and each line executed is added to the trace which can be displayed in a text output. This has a stepped feature that indicates the scope that the program is operating within (shown on the right hand side of Figure 5.16).

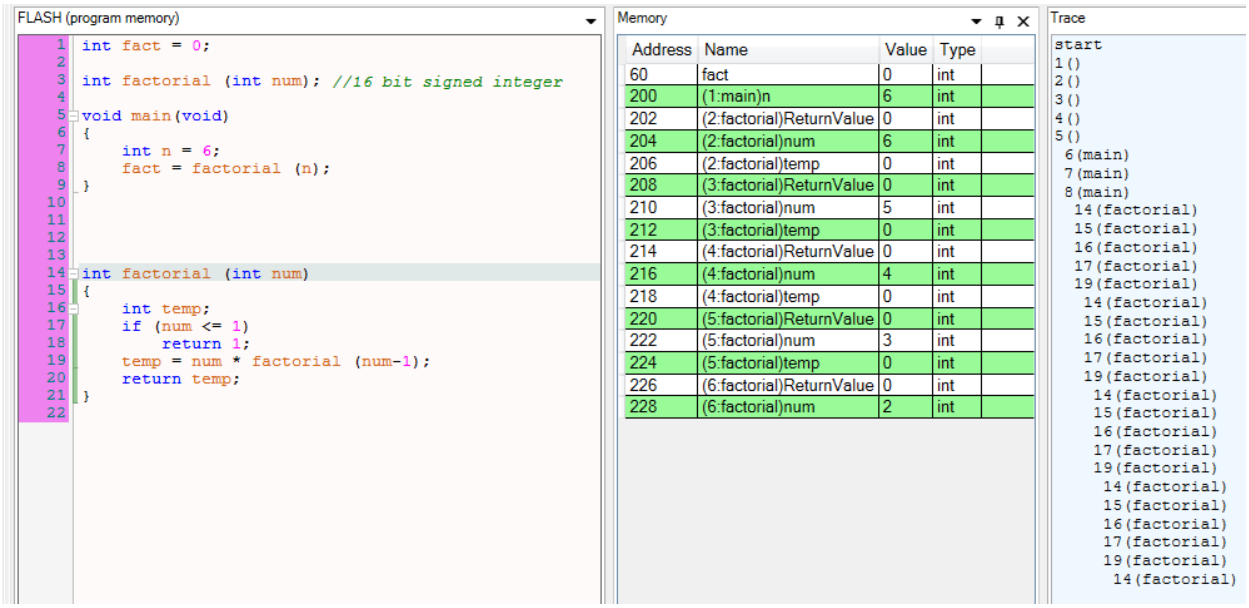


Figure 5.16: Visualiser with trace and stepped scope view

Program code entered into the visualiser is stored between sessions to model the nature of FLASH memory. EEPROM memory storage is currently not implemented but will also be stored between sessions when the code for that is completed.

Currently requirements for semicolons at the end of lines of C program code have been removed from the visualiser. An automatic insertion of these will be implemented as time allows; the

reason for this is to focus students away from the common error of leaving out semicolons because they don't know where to put them.

An issue that modern instructional methodology should not overlook (Ertmer & Newby, 2013) is collegial work practices. Each separate diagram in System Designer is saved as its own file, so a project that has 10 drawings will have 11 files (the 11th file maintains the list of files for the full project). As students move through various phases of their projects they will work collaboratively in groups, sometimes moving between groups as well. Groups can easily share individual files with each other simply by copying the file for a particular diagram into their project folder. When System Designer is opened it will ask if the file should be added to the project. To avoid plagiarism issues all files are tagged with the name of the person who created the project where the file was first created. If a student presented work where few of the files were created by them, there is a method of tracing authenticity. This means that students can collaborate together yet assessment practices maintain integrity.

5.5. Extensibility

Even though developers put much thought into the development of visualisation tools the worth of a tool is measured by educators based upon its ease of use and their "feeling of control" over it (Levy & Ben-Ari, 2008, p. 172). This means making the visualiser as flexible and extensible as possible. To make it as compatible with varying learning programmes as possible and with the unknown needs of other educators, many of its features have been implemented by having the tool read certain directories and various machine readable XML files on start-up. For example to add a new AVR microcontroller or input device, the appropriate XML file can be copied from an existing one and edited and a new appropriately named image added to the correct directory. System Designer will read these on start-up and integrate them into its menu system automatically.

5.6. Summary

Prior to this thesis System Designer had grown in a reactive and somewhat ad hoc way to meet students' needs for planning and developing their projects. An intuitive, but vague, awareness of the direction needed for its future growth had been developing; this research provided the vital time and crucial separation from the classroom needed to undertake the deep analysis required to direct its development; as well as this it afforded the added benefit of viewing embedded systems learning at the tertiary level. It also provided the essential opportunity to reflect upon the existing

capabilities and limitations of System Designer in terms of visualisation literature. The investigation of existing visualisation environments, their analysis against educational literature about model-based learning and the subsequent abstraction of their salient features provided the essential pedagogical direction for the visualiser. This research further provided the opportunity to investigate the more advanced programming techniques required to rework the existing application and extend it to become a dynamic visualisation tool, while at the same time considering other educators' needs to adapt the tool for their own use.

The visualiser has been developed as a learning tool and not as a replacement for the important learning that takes place when using real hardware; it will be used in conjunction with real hardware so that students learn about embedded systems and not the tool itself. Also the visualiser was not envisaged as an embedded systems development environment; so some features such as the integration with compilers and some aspects of C and BASIC languages have not been included, so that the visualiser would stay true to its core purpose as a learning tool.

Chapter 6. Visualiser support materials development

Teachers ultimately maintain control over what will take place in their classrooms. Consequently a new pedagogical understanding or tool such as a visualiser, even though it is well founded in theory and research, will always face the scrutiny of educators who balance many demands for their attention. One of the reasons visualisation tools are not widely used is that their developers often do not take the time, or they do not have the background required, to support educators integrating the tools with existing learning programmes (Levy & Ben-Ari, 2007). The material presented in this chapter is the crucial synthesis of the new understandings gained from literature about visualisation, embedded systems and learning to program with the tasks already used in class to facilitate deeper conceptual understandings. The chapter includes the important content that must be covered when learning about embedded systems and programming (section 6.1) and then presents a discussion of how the existing classroom exercises and materials have been extended or changed to incorporate the powerful new learning opportunities offered by the visualiser (section 6.2).

6.1. Content coverage

Our responsibility is to pre-empt the naïve and strange understandings of students by developing quality resources to expose, identify and replace them. This requires a focus on the important aspects of learning in embedded systems, visualisation, learning to program and pedagogy.

6.1.1. Embedded systems

The crucial relationship between hardware and software experienced in the classroom is best supported by student interaction with real hardware; the visualiser will not replace this but supplement it by exposing many of the hidden operations and interactions of embedded systems in a dynamic way. The visualiser support materials will need to build understandings of a wide range of peripherals and their interactions with microcontroller hardware.

There should be a developing awareness of the real time nature of embedded systems and synchrony. Tasks should capture aspects such as the need for synchronisation with real world interfaces; this can be introduced with understandings around the length of press of a button as being very slow compared to the microcontroller clock and the associated contact bounce and

debouncing of switches in software. Polling versus interrupt driven methods of programming will need to be supported so that students can overcome synchronisation issues.

6.1.2. Visualisation

Support materials for the visualiser should leverage off the range of understandings known already about both the benefits and limitations of visualisation. Educators need to be highly explicit about how and what information students should take from resource materials (Alton-Lee, 2003) so pre-training them on using the visualiser is essential (Kaila, Rajala, Laakso, & Salakoski, 2011; Sorva, 2013) so that when they use it their attention is fully on its cognitive aspects (Mayer & Moreno, 2003). Visualisations are analogies and are easily misinterpreted. To avoid this educators need to be explicit on the meaning that should be taken from visualisation use (Clancy, 2004). Perhaps the most significant characteristic known about visualisation is that the powerful benefits of it are only realised when students change, create or present their own visualisations and not through simple demonstration (Coll et al., 2005; Naps et al., 2003).

6.1.3. Programming

A wide range of understandings that exist about learning to program need to be taken into account when building tasks for learners. Students make assumptions about what program code is actually doing (Putnam, Sleeman, Baxter, & Kuspa, 1986; Sleeman, Putnam, Baxter, & Kuspa, 1986), often attaching intelligence or some hidden mind to a program (Rogalski & Samurcay, 1990) rather than a more accurate obedience model. Students need code tracing and comprehension exercises to unlock this meaning (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Rountree, Rountree, Robins, & Hannah, 2005; Sudol-DeLyser, Stehlik, & Carver, 2012) especially around method calling and execution (Ahmadzadeh, Elliman, & Higgins, 2005; Goldman et al., 2008).

Exercises should not just build schemas (reusable blocks) as novices consistently use them incorrectly (Spohrer & Soloway, 1986; Winslow, 1996). It is crucial to focus on the relational understandings and strategies for using code structures (de Raadt, Watson, & Toleman, 2009b; de Raadt, 2008a, 2008b; Hundhausen et al., 2002; Rist, 1989). Schema creation is enhanced by separating out the various roles of variables in programs (Al-Barakati & Al-Aama, 2009; Ben-Ari & Sajaniemi, 2004; Byckling & Sajaniemi, 2006; Kuittinen & Sajaniemi, 2004; Sajaniemi, 2002).

Programming examples should be not cluttered with extra information (Malan & Halland, 2004) and complex lines of terse code should be unravelled into individual concepts (Bennedsen &

Caspersen, 2008). Explicit teaching of debugging skills and how to unpack error messages is essential (Ahmadzadeh et al., 2005; Coull & Duncan, 2011; Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011; Flowers, Carver, & Jackson, 2004; McCauley et al., 2008) with focus not on the error but the problem the message reflects (Spohrer & Soloway, 1986).

Memory model and understandings of variables are troublesome for students. This includes type related problems (Ahmadzadeh et al., 2005; Denny et al., 2011; Goldman et al., 2008) and assignment (Kaczmarczyk et al., 2010), as well as parameter passing using pointers (Adcock et al., 2007; Ahmadzadeh et al., 2005; Goldman et al., 2008) and issues with array subscripts (du Boulay, 1986; Kaczmarczyk et al., 2010).

Keywords used in programming such as: 'while', 'do', 'for' and 'if' breed significant misunderstandings because of their English language meanings that do not directly relate to their use in programming so they must be explicitly unpacked for novices (Clancy, 2004; du Boulay, 1986; Goldman et al., 2008; Putnam et al., 1986). The conditional 'if' can be particularly challenging as it shares Boolean logic with looping, leading even senior university students to experience troubles with their "if-loops" (P. Roop, personal communication, 4 September 2013).

Looping constructs and the mix of control variables as part of Boolean tests are highly problematic (Goldman et al., 2008; Kaczmarczyk et al., 2010; Krone et al., 2010; Pea, 1986; Putnam et al., 1986; Rogalski & Samurcay, 1990; Yamamoto, Sekiya, & Yamaguchi, 2011), especially the 'off-by-one bug' (McCauley et al., 2008; Spohrer & Soloway, 1986) and the failure of a loop to terminate (Simon et al., 2007).

Use of a wide range of real contextualised example programs as problem exercises rather than decontextualised concepts supports deeper levels of understanding when learning to program (Guzdial, 2010; Lahtinen, Ahoniemi, et al., 2007; Malan & Halland, 2004; Van Merriënboer & Paas, 1990). This helps move education away from a telling model about programming structures to a 'selling' model where benefits are expounded thus motivating students to use them (Malan & Halland, 2004).

6.1.4. Application of pedagogy

Understanding of various learning theories has always played a crucial role in developing classroom resources. At the centre of classroom work is the selection of strategies; from behavioural training to cognitive development to constructivist guiding to building participatory learners. Modification of resources must be congruent with the curriculum goal of building self-

directed learners so simple rote learning of technical detail should not replace the building of conceptual understandings.

Choosing how a learning object such as a task delivers its content is important in achieving this. The material must clearly direct the student to focus on the required understanding; so the teacher must "pay close attention to what varies and what is invariant" (Thuné & Eckerdal, 2009, p. 344). A learning object should do this by presenting the variation between the concept of interest and other concepts; four patterns for achieving this are contrast, generalisation, separation and fusion (Marton, Tsui, & Runesson, 2004).

The contextualisation of exercises has always been a crucial component of classroom work. It links students to the real world, as required in the curriculum; it begins the understandings of situation and stakeholder needs which are the basis for requirements engineering and technology education; it builds awareness of the essential aspect of fitness for purpose in embedded systems as this is entirely context related; it supports students thinking about deeper understandings of computer programs (Thuné & Eckerdal, 2009) and it promotes deeper thinking rather than behaviourist patterns of learning. Using a range of contexts that students are familiar with will support learning without introducing extraneous processing. Contexts previously used with students will continue to be relevant, these include: a stopwatch, timers for various applications, dishwasher, electronic sign, washing machine, heat-pump, escalator, game contestant buzzer, pedestrian crossing, tachometer, heart rate monitor, scoreboard, ticket machine and garden sprinkler controller.

6.2. Tasks development and integration

The exercises described here have been previously developed as part of the first course that students take at school. These are year 10 (14-15 year old) students and the course is two school terms. This equates to 17 weeks of learning time with four 50 minute lessons per week. These begin with simple instruction and sequence exercises and build progressively to include variable use and repetition and functions as part of project based learning exercises. The instruction and sequence exercises are discussed, along with how they have been modified by integrating the visualiser into them. The second set of exercises has been significantly extended from those currently in use; this reflects the growing importance in this study of making strategies and plans explicit to novices. Further exercises are under development for use with school students and will form part of future research.

6.2.1. First task set: instruction, sequence and state related tasks

The first tasks used with students in class focus on understanding simple microcontroller interfaces using LEDs and switches. These are used to support initial understandings of instructions, sequence, state and flow of program code initially in very simple programs progressing through to programs using several functions.

- Task A is to flash a single LED (simple instructions to control bits in a port).
- Task B is to develop a light sequencer using eight LEDs (surface understanding of sequence and state).
- Task C is to develop an LED sequencer for 15 LEDs (deep understanding of sequence).
- Task D is a controlled pedestrian crossing sequence exercise (sequence of program code in a more typical context).
- Task E is a Morse code exercise (builds on sequence and state, and introduces code reuse using functions).

A direct instruction lesson introduces students to the textbook (Collis, 2013), Task A and Task B. Further whole class learning time is used to review the exercises and to review learning intentions, not to preload students with information about new exercises. Students instead are referred to the textbook for further instructions as they progress with each task.

In the short direct instruction lesson Task A and Task B are introduced with one LED flashing then three LEDs flashing in a sequence. From there, students are instructed to progressively add more LEDs to mimic the LED sequencer for the car ‘KITT’ from the TV show Knightrider. Figure 6.1 shows the classroom demonstration system. (Note that all students have assembled and soldered their own hardware to experiment with prior to these exercises.) This exercise is covered in section 6.19 of the textbook. These tasks are designed to build student familiarity with sequence as well as with the IDE and the programming process.

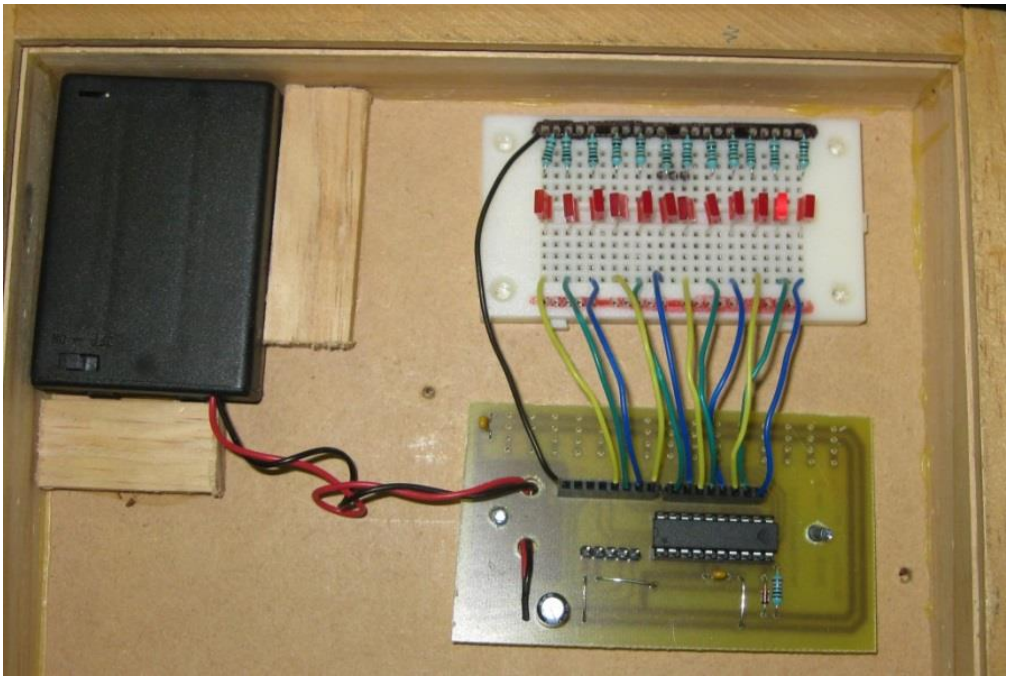


Figure 6.1: Classroom demonstration system of a 12 LED sequencer

With up to eight LEDs in Task B, all can be attached to a single microcontroller port and are easily controlled with single instructions. In class almost all students very quickly pick up the code pattern that creates this sequence. A portion of the typical pattern of code that would be generated by a student is shown below

```
Students code for led sequence
PORTA = 0b00001000
Delay…
PORTA = 0b00010000
Delay…
PORTA = 0b00100000
Delay…
PORTA = 0b01000000
Delay…
PORTA = 0b10000000
```

In Task C students consistently encountered significant challenge when more than eight LEDs are added to the sequence. This is related to the fact that students must add LEDs to a second microcontroller port. Over time a number of student errors have been catalogued. These are: forgetting to add the code for the new LEDs and assuming they will work automatically, copying and pasting the sequence for one port to another without realising that the sequence must go backwards (because the pins are in reverse order) and adding code for non-existent PORTB.7. Each of these is encountered with every new cohort of students in the course and feature valuable learning moments for them.

One issue in particular has proved to be very challenging for most students. In the code below, while the student has added the correct command for the LED on PORTB, the state of the last LED on PORTA however has not been changed from on to off. Du Boulay (1981) identified the two concepts novices need to develop that are highlighted by this situation as sequential (what a program is doing) and circumstantial (what came before the current state).

```
Students incorrect code for more than 8 LEDs
PORTA = 0b01000000
Delay…
PORTA = 0b10000000 - student does not realise they need to turn off this LED, it stays on
Delay…
PORTB = 0b01000000  - student has correctly added a command to control PORTB.6 LED
Delay…
PORTB = 0b00100000
Delay…
PORTB = 0b00100000
```

The student needs to add a second line of code for the transition between PORTA LEDs and PORTB LEDS to turn off the LED on PORTA.7 as per the code below. Students often struggle with this understanding. In doing this they present an 'egocentric' model of program understanding (McCauley et al., 2008) as they expect the microcontroller to know for itself what it should do and often attribute more to a command than it is actually capable of.

```
The correct code is
PORTA = 0b01000000
Delay…
PORTA = 0b10000000
Delay…
PORTA = 0b00000000   - turn off the LED on PORTA
PORTB = 0b01000000   - turn on the LED on  PORTB
Delay…
PORTB = 0b00100000
```

A further and common outcome of this is for students to add an extra delay in the sequence thus causing a glitch or jerkiness to the pattern. In this they are presenting a rote-learning or behaviourist pattern to their learning, where they have been able to correctly follow the pattern but are confused when tackling something that requires genuine understanding.

```
PORTA = 0b10000000
Delay…
```

```
PORTA = 0b00000000   - turn off the LED on PORTA
Delay…               -added incorrect extra delay
PORTB = 0b01000000   - turn on the LED on  PORTB
Delay…
PORTB = 0b00100000
```

At any of these stages many students are generally unable to correctly identify what the problem is. A number of students never identify the problem themselves and resort to copying a friend's program, even then revealing no or little understanding when questioned.

Questioning of groups of students when seeking to guide them typically starts by having students clarify what a specific line of code does e.g. PORTA = 0b00001000. A student will often express it partially correctly by saying it is turning on the 4th LED; a comprehensive understanding however is that it turns off any LEDs on the port that may be currently on, except for the 4th LED which it turns on (or leaves it turned on if already on). The majority of students can appreciate this when it is explained to them however they are still unable to make the connection to how to solve their problem with the LED remaining on. Whilst no specific data has been captured this is a recurring pattern with each cohort of students.

6.2.1.1. Prior pedagogical context of the LED sequence exercise

The very difficult aspect in Task C of transitioning beyond 8 LEDs is strategically not covered by the prior direct instruction given to students; instead this aspect is used to develop an awareness of students' capabilities as learners. Up to this point in the class the year 10 students have had four or five weeks where their learning has been managed for them. A few short direct instruction lessons have been given but mostly practical tasks are used in class. During this time I have had the opportunity to train students behaviourally in responsible and collegial workshop practices and build their awareness of attention to detail of such things as component recognition and polarity issues, accuracy of soldering, and syntax through copying programs into the program editor. The building of attention to detail is a precursor to switching from a task to a process level of feedback. During this time the instructions given to students will slowly change from telling students what is wrong to encouraging students to methodically check their own work and identify sources of problems for themselves based upon what they know already. During this time I have had the opportunity to discover their willingness to comply with instructions, their fine motor skill capabilities, and also to begin to appreciate the individuals and any special learning requirements that some students may have.

Task C is the first of many challenges students will encounter where they are faced with a problem that cannot be overcome with rote-learned behaviour. It is essential that they move beyond behavioural models of learning to cognitive understandings, so in this exercise they will not be given any task related feedback. Their learning will be heightened by process related feedback to help them identify the areas in which they need to focus and they will be encouraged that a certain level of struggle is essential to gaining real understanding. This is made explicit to students who find this process difficult so that they become aware of the expectations being made on them. A formative assessment is made of students at this time as to their power as learners. This is to fulfil the goals set by the New Zealand Curriculum and relate to the student's capacity, resilience and responsibility for their own learning.

At this time the more powerful learners in the class will be clearly identified and they are encouraged to become more self-directed and set off at their own pace with further tasks. Other students are questioned to identify where their understandings are at, and in small groups are supported as to where they might find support and given clues or more specific directions in which to set out to find help. This is the model for all project and problem based learning in the classroom and essential for building resilience in students. Nonetheless there are a few students that are highly resistant to this model of instruction and sometimes will cease work output on Task C. The aims of self-directed learning will not be pursued further with these students at this time. Instead they are directed onto the next tasks so as to keep up momentum with the rest of the class and continually encouraged to move beyond the model of thinking they are comfortable with.

6.2.1.2. Introduction of the visualiser to student learning

Introducing a new learning tool into an existing situation requires a clear understanding of the dynamics of the existing course and perspective on how it might alter them. The planned introduction of the visualiser is not to use it as a demonstration tool for difficult issues such as in Task C, as research literature into visualisation techniques has already identified demonstration effects as low for student learning. Students must actively engage with the tool and so it will be introduced as a potential strategy they can use to help them solve the problems they are facing with their project based learning tasks.

It is anticipated that all students will find the use of the visualiser as motivating because of the responsive nature of the feedback it provides compared to the use of actual hardware. The

visualiser however must not be allowed to completely replace the use of actual hardware and negate the benefits that real hardware provides.

Task D and Task E depart from simple sequence examples. In the class there has typically been a general reluctance on the part of many students to move beyond the 'comfortable' simple flashing LED exercises; on occasion this requires actual removal of LEDs from students.

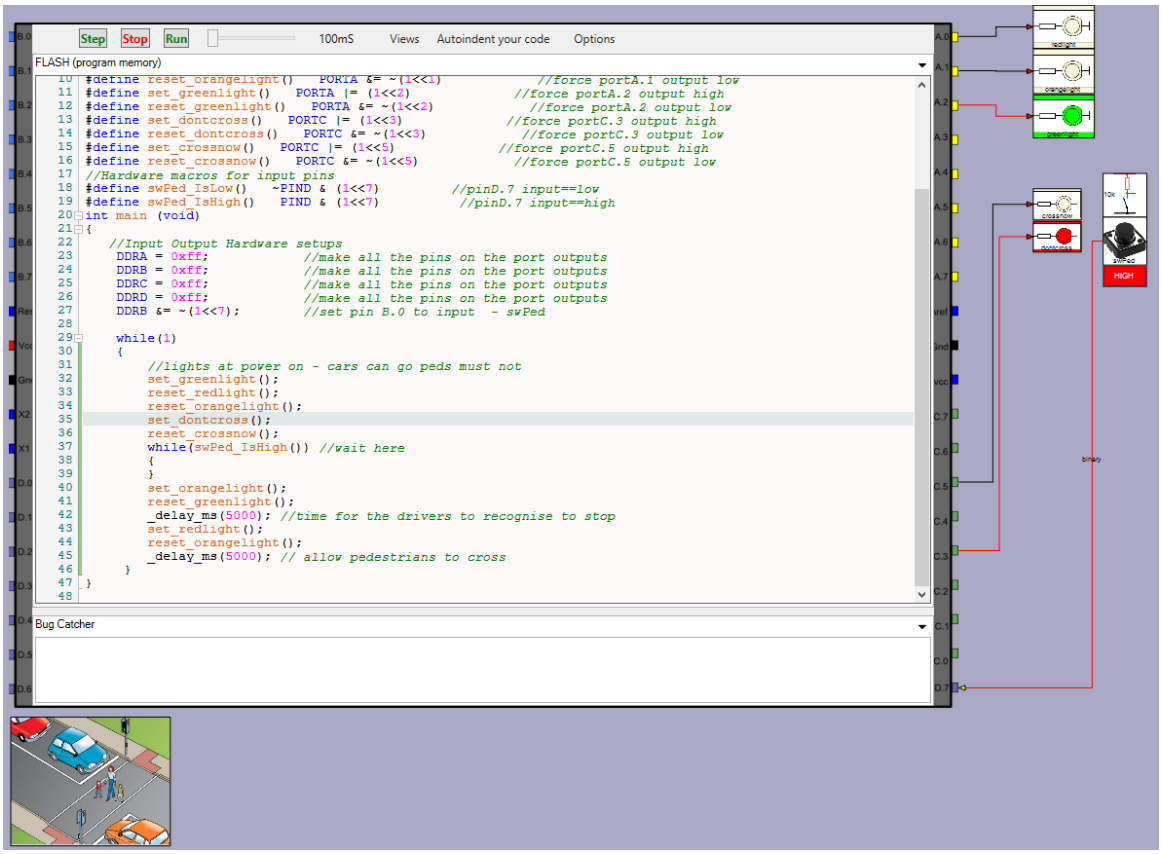


Figure 6.2: Sequence task D – controlled pedestrian crossing

Task D requires students to work with a sequence of disparate LED objects, specifically the lights in a controlled pedestrian crossing. It also introduces the first use of input control (via a push button switch). Over time a range of static strategies have been trialled with students to develop their thinking, including tables and flowcharts as in the textbook (Collis, 2013, p. 115). Many students however still struggle to implement the sequence. In the future the task will be introduced to students using hardware (as it is currently), however students will then be instructed to change to using the visualiser to complete the program for the lights using the template in Figure 6.2 before rolling it out onto their own hardware. This includes modification and

extension exercises that require students to carry out research into an actual pedestrian crossing to fully appreciate the timing and sequence of a real set of lights.

Task E is a Morse code exercise. Previously students programmed their system to output their name in Morse code using an LED and a piezo sound output. The understanding being developed is around the use of functions for program code reuse. This exercise has been strengthened contextually using a real world application of Morse code in VOR (VHF Omnidirectional Radio Ranging) systems as part of a runway approach as in Figure 6.3. Whether it is better for the students to build the hardware and use the visualiser alongside it or only use the visualiser will be trialled in the classroom to see which scenario ultimately proves to be more powerful for student learning. It is anticipated that even if students still construct the hardware this exercise will take considerably less time than the current hardware-only exercise because of the increased benefits of visualisation for understanding.

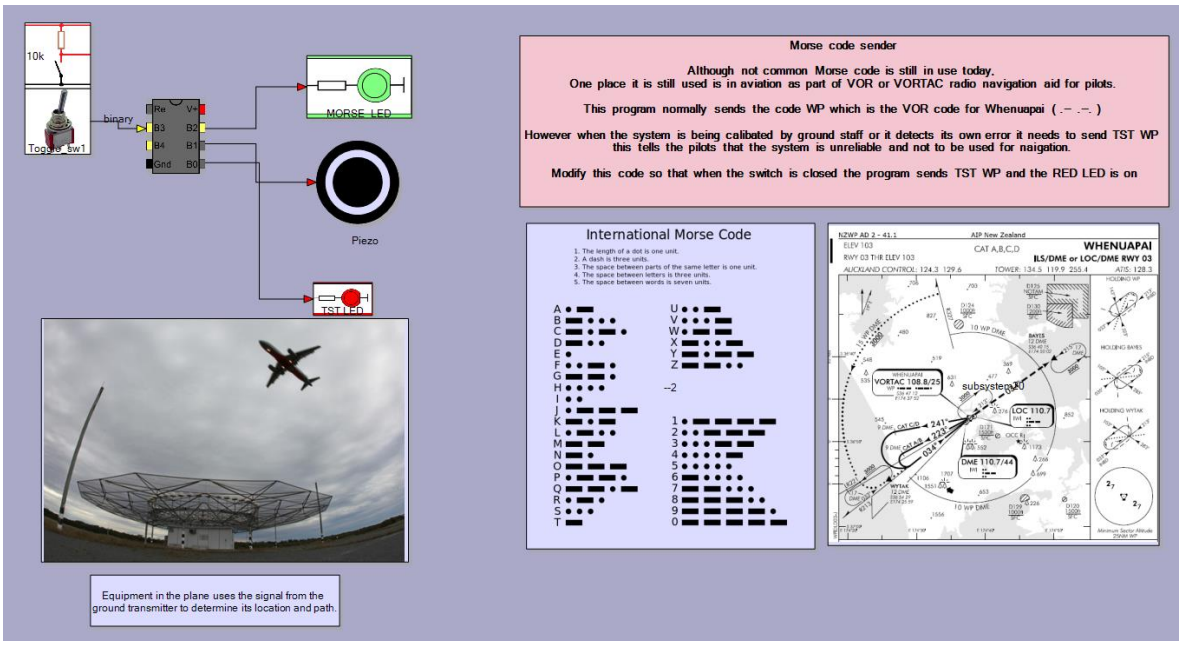


Figure 6.3: Morse code exercise exemplifying program reuse.

In class there will always be a range in the time taken for different students to develop understanding. Students who finish earlier are usually given extension exercises; these will now be replaced by a Dragway lights sequencer as shown in Figure 6.4 as the visualiser presents the opportunity for new contexts without the lengthy hardware building process.

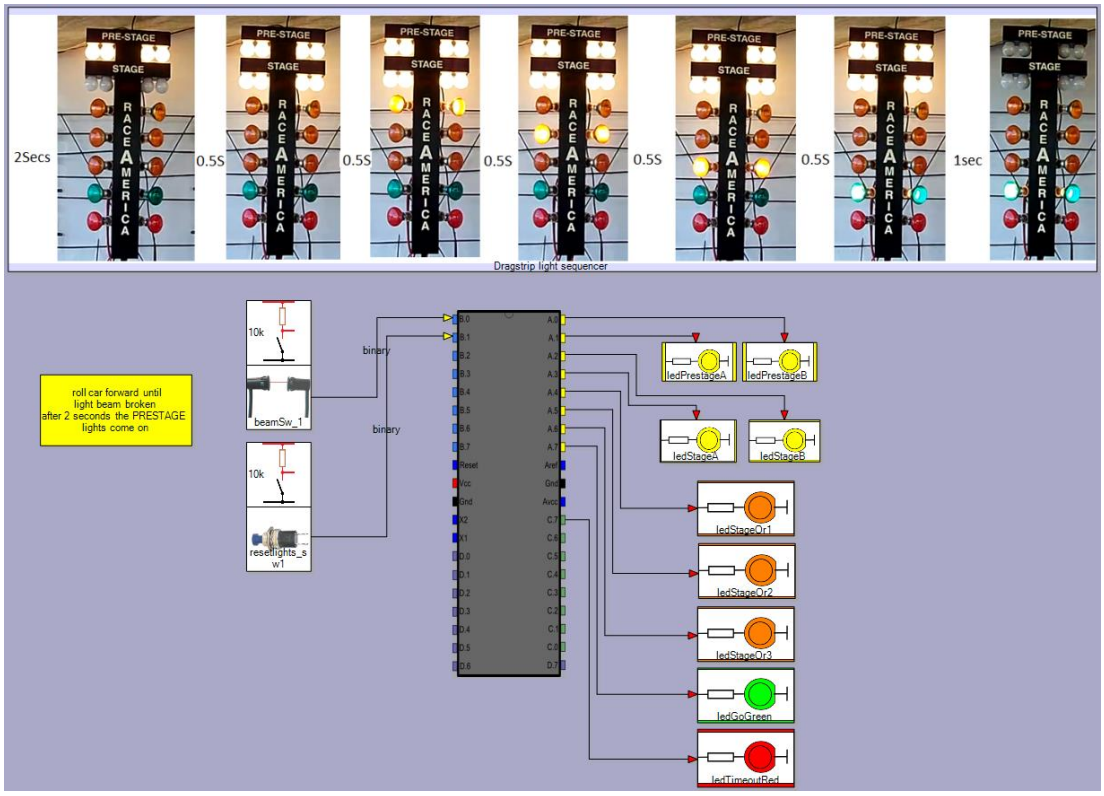


Figure 6.4: Dragway lights sequencer for early finishing students

6.2.2. Strategic knowledge and relational understandings

Prior classroom work on making strategic knowledge explicit has centred on state machines in the year 12 and year 13 courses. Literature encountered through this research however strongly indicates that there is a need for more focus on lower level strategies with students earlier in their learning at the sub-algorithmic level (de Raadt et al., 2009b); such as with repetition and loop control, counting and variable swapping. This is similar to the pre-algorithmic work on 'roles of variables' (Ben-Ari & Sajaniemi, 2004; Kuittinen & Sajaniemi, 2004; Sajaniemi & Kuittinen, 2003; Sajaniemi, 2002; Sorva, Karavirta, & Korhonen, 2007).

Both the teaching of strategies and roles of variables recognise the importance for students in crossing the boundary from declarative to know-how knowledge, from structure to function (Schulte, 2008) and from surface to deep understandings (Robins et al., 2003; Thuné & Eckerdal, 2009). To facilitate students' learning Soloway (1986) identified four strategies for 'gluing' program plans together: abutment (or sequential), nesting, merging and tailoring (altering a canned plan) and de Raadt makes extensive reference to these in his Programming Strategies Reference (de Raadt, 2008a).

To undertake an initial investigation of explicit teaching of strategy the 'backyard alarm' task has been redeveloped for this study using the flow chart features of System Designer. Figure 6.5 shows the block diagram developed for the exercise.

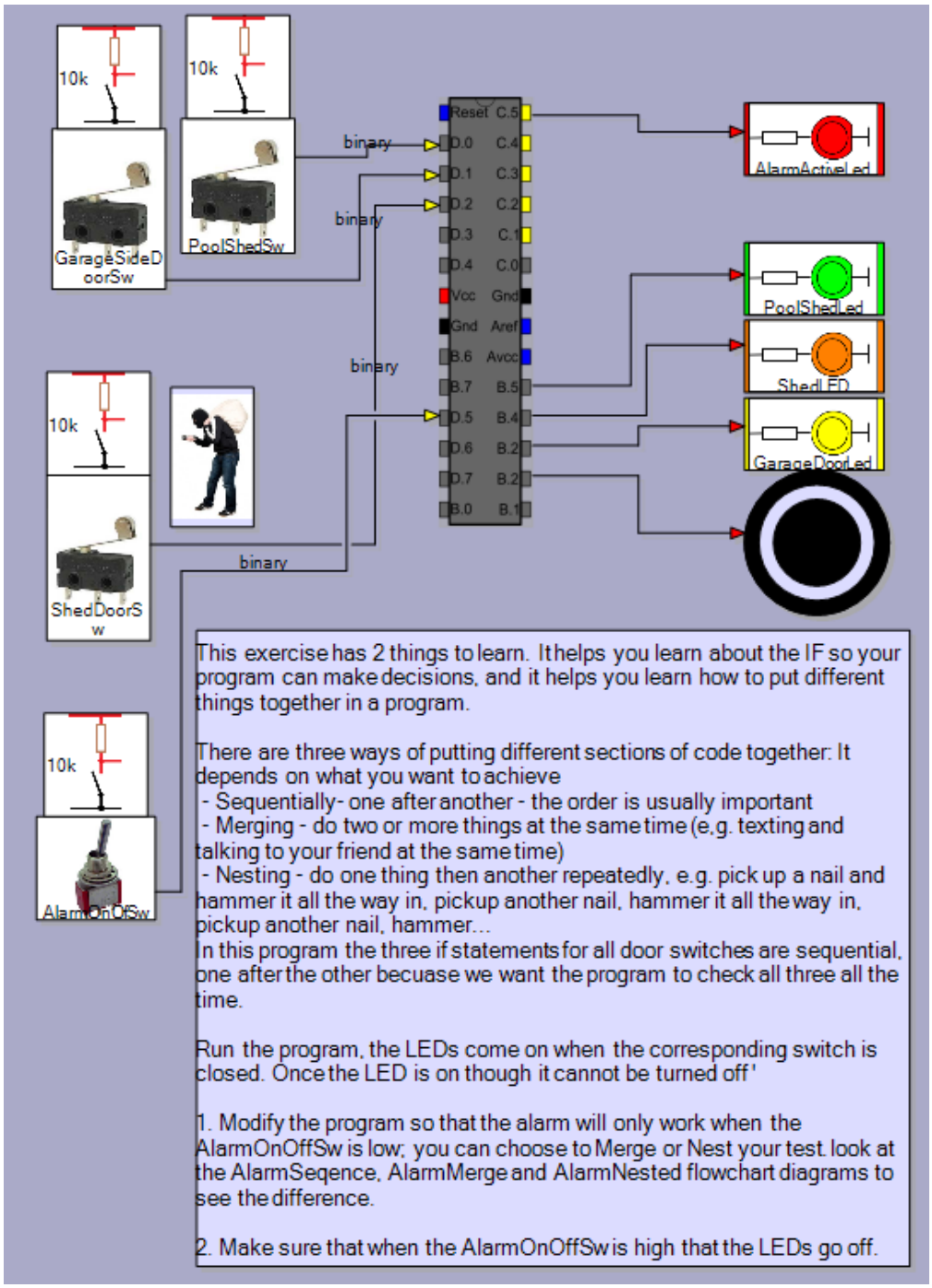


Figure 6.5: 'If' with merge and nest strategies

Figure 6.6 shows System Designer's flowchart editor. It has three sections: a drag and drop flowchart on the left, a table of inputs, outputs and variables in the centre that is shared from the block diagram and a code window on the right. The flowchart shows three sequential if-statements forming tests for the three door switches in the block diagram of Figure 6.5. This

editor has a drag and drop interface for common programming constructs including: input-output command, process command, for-loop, do-while loop, while-loop, if-test and if-else-test. The application automatically produces program code for these as they are dropped into the flowchart. A drop down menu (right click on a flowchart item) has code hints in it which are generated from the input, output and variables tables. Students can copy and paste this code into their own program editor or into the visualiser. When using a task such as this the student's attention will need to be drawn to the sequential nature of the flow and the students questioned on whether order is important or not in this particular context.

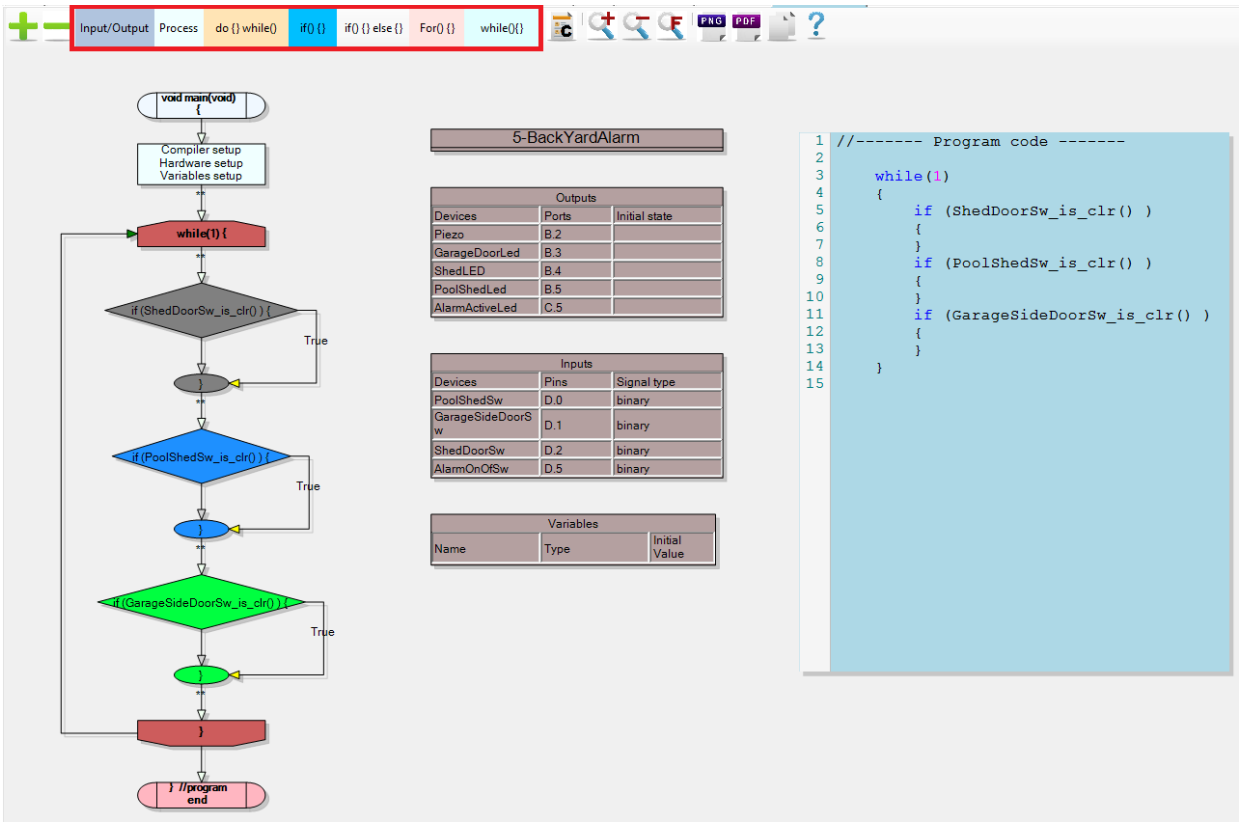


Figure 6.6: System Designer flowchart editor with backyard alarm sequence

Figure 6.7 shows the visualiser open with the program code for the backyard alarm already working. When one of the three switches is closed the program turns on the matching LED. There is, however, no mechanism to turn off the LEDs. The students are required to add an override switch for the alarm that will allow users to set and reset the three trigger switches all at once. This requires students to make use of either a nest or merge strategy. Examples of each strategy are made available for students as shown in the example flowcharts of Figure 6.8 and Figure 6.9. Students will be instructed to drag and drop blocks into the flowchart to build a strategy for the overall system, before testing it in the visualiser.

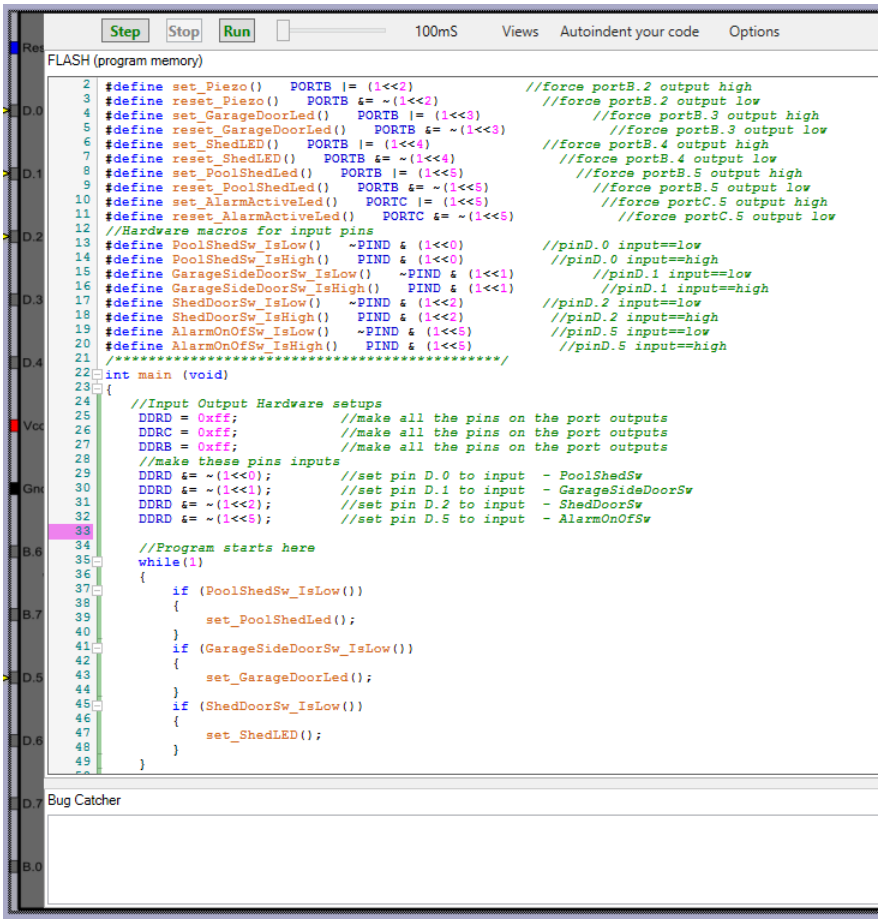


Figure 6.7: Backyard alarm starter code in the visualiser

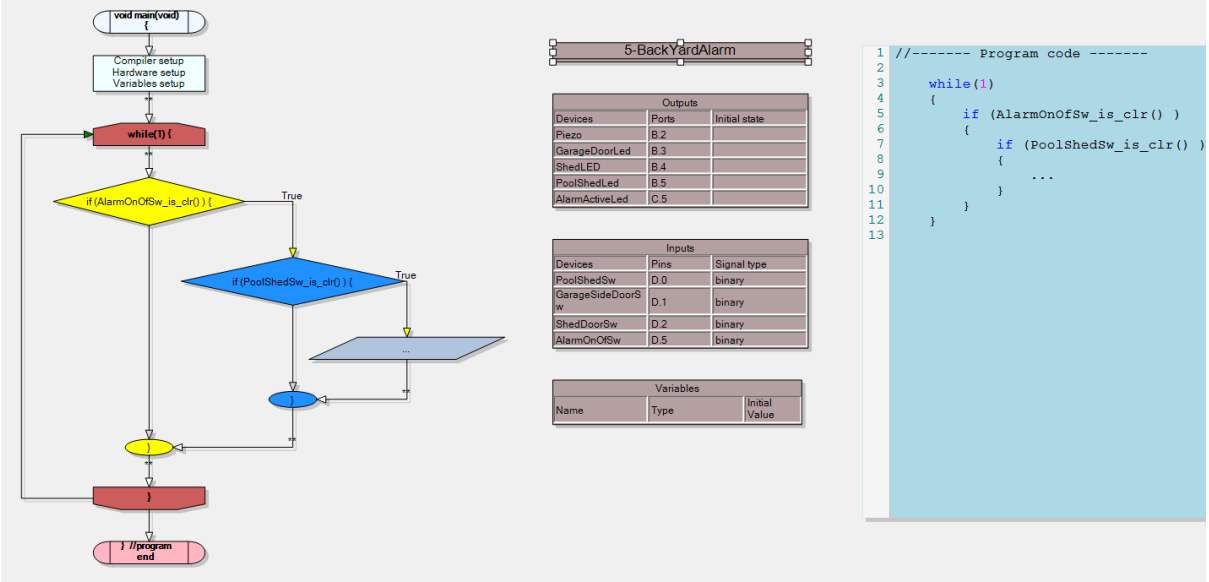


Figure 6.8: 'If' with nest strategy

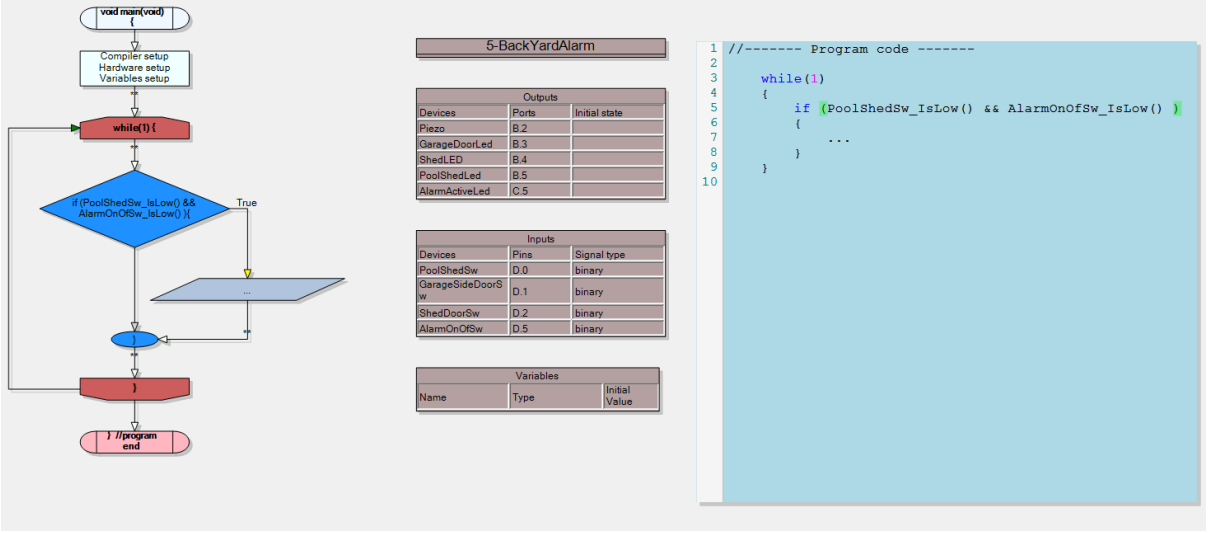


Figure 6.9: 'If' with merge strategy

Figure 6.10 is a second similar task for students on merge and nest strategy development; this is a game show style contestant situation with four buttons and a reset button. Students will be instructed to create a flowchart for the program themselves.

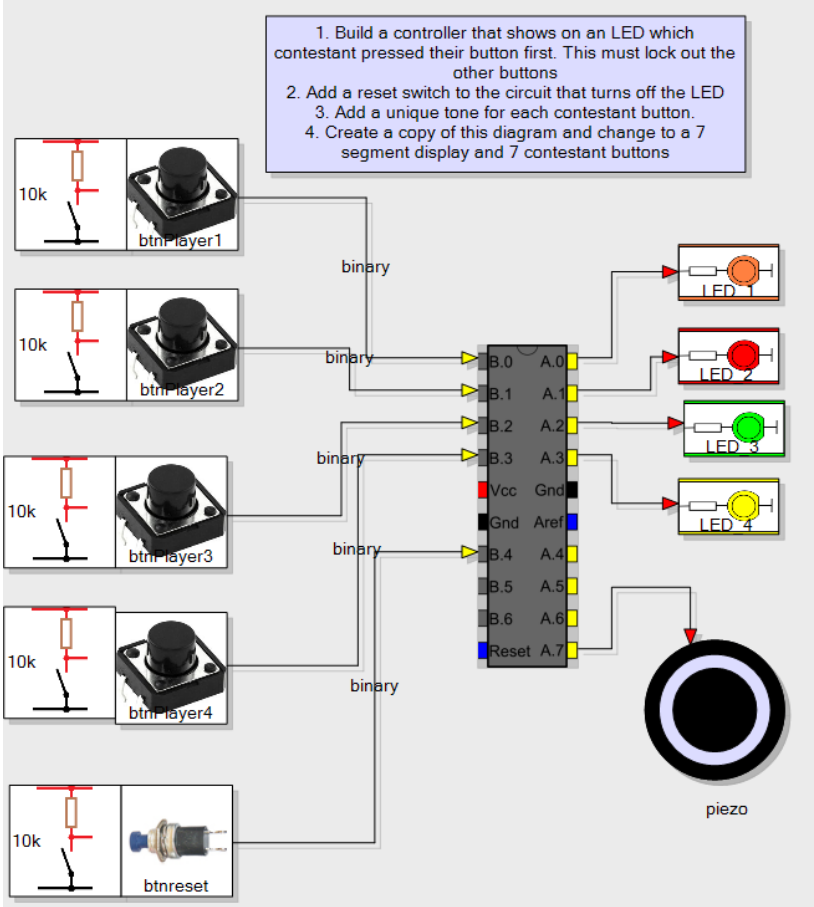


Figure 6.10: Programming plans merge and nest exercise

6.2.2.1. Second task set: variables

Four exercises used with the year 10 students to explicitly build recognition of roles of variables use in programs are shown here: a dice program in Figure 6.11, a light meter in Figure 6.12, a door timer in Figure 6.13 and a temperature comparison system in Figure 6.14. At this level students' use of hardware is constrained to a range of switches, simple analog sensors (light dependent resistor and LM35 temperature sensor), LEDs and seven segment displays. Rather than having students do only one practical task, as they have previously, the visualiser offers the opportunity to carry out all four exercises; students will then complete one of these as a practical task using actual hardware.

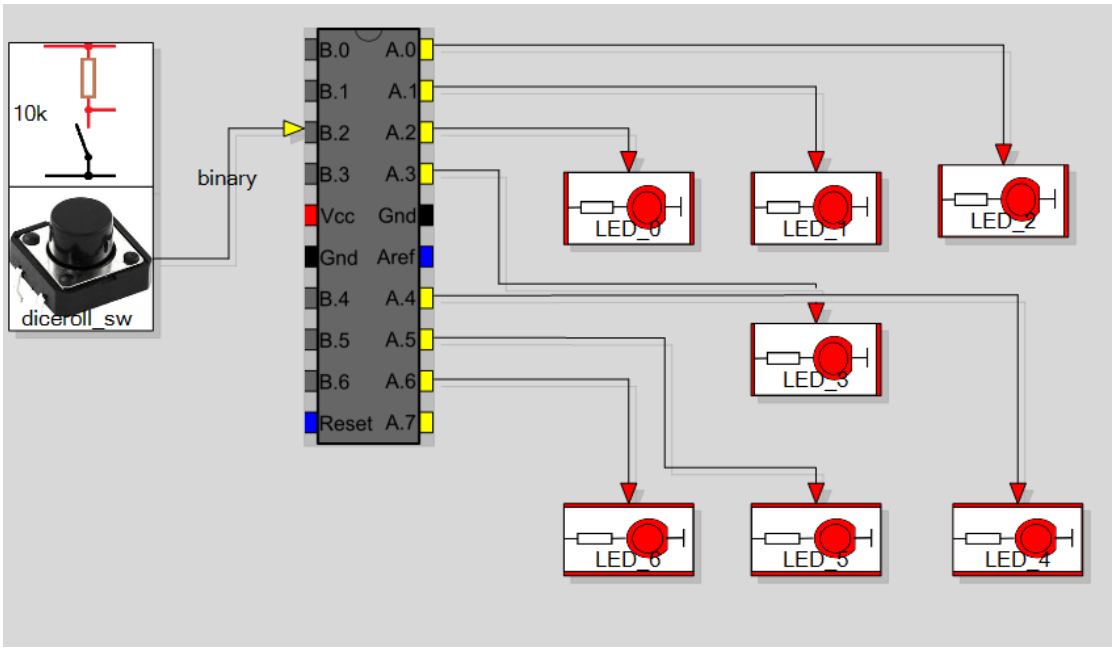


Figure 6.11: Dice program (year 10) - ROV (role of variable) as stepper

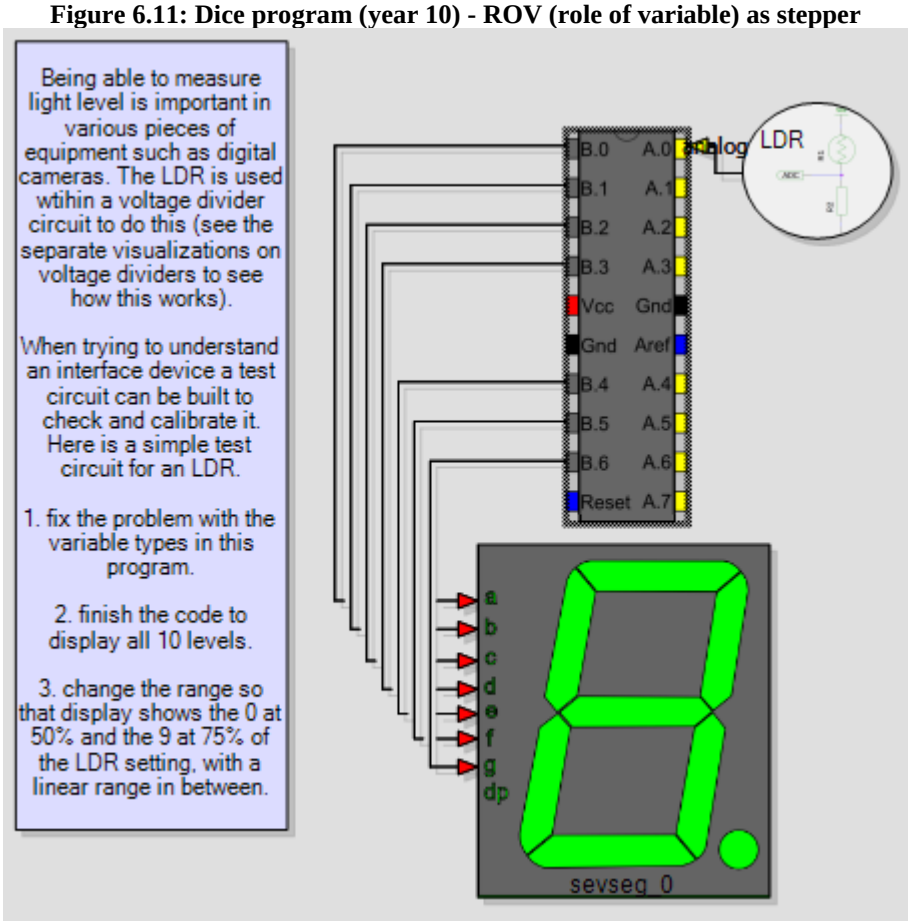


Figure 6.12: Light meter (year 10) - ROV as most-recent holder

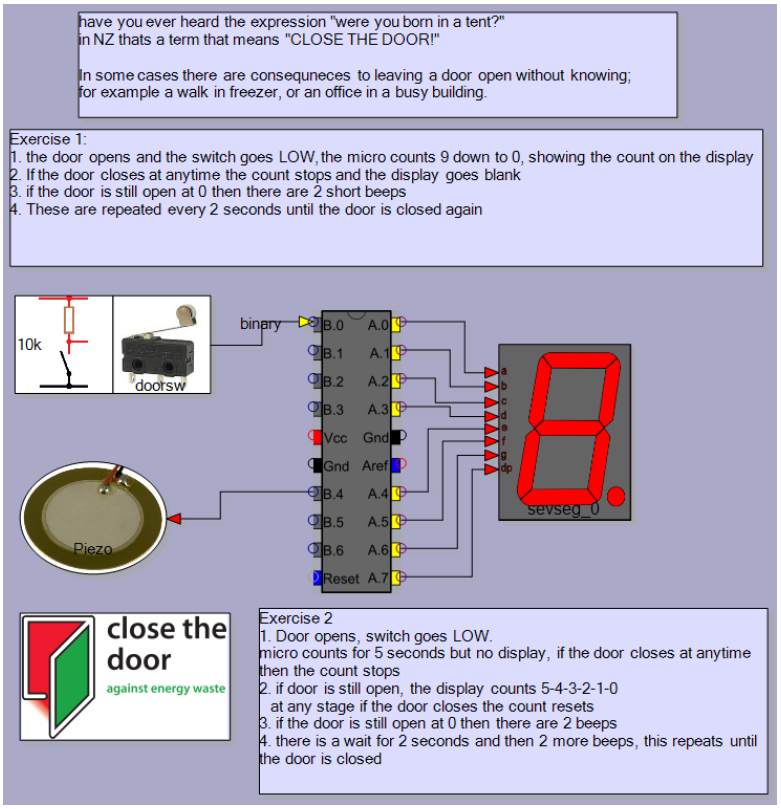


Figure 6.13: Door time open counter (year10) - ROV as most recent holder

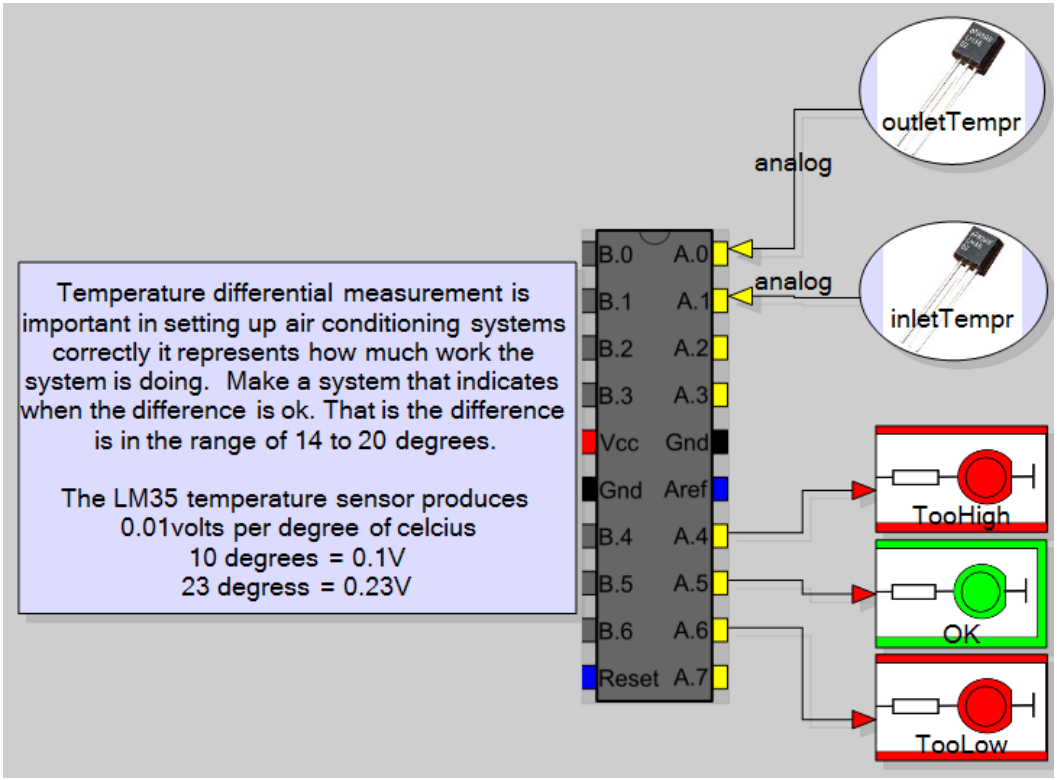


Figure 6.14: Temperature comparison system - ROV as most wanted holder)

6.3. Summary of task development

Development of tasks and exercises for student use is based upon a teacher's deep 'pedagogical content knowledge' (Shulman, 1987) which is a wide range of curriculum, subject, pedagogical and content understandings. The tasks are a culmination of an educator's thinking about the learning students face and the consequent selection of the best strategies to support students out of misconceptions and toward viable conceptual understandings. These strategies require students to have sufficient opportunities to learn and require timely and formative feedback related to the task, process or thinking students are required to develop. The strategies also leverage off various learning theories from behavioural training to cognitive guidance to the creation of situations in which students construct their own meaning. The visualiser should not be seen as replacing existing pedagogical content knowledge but deepening and broadening it. Consequently the exercises discussed in this chapter are not new, they have been taken from the existing programme of learning. They have however been modified and extended through reflection on the new understandings gained from research into visualisation. They will require ongoing testing and honing, and possibly replacing as their use reveals their strengths and weaknesses.

Chapter 7. Research methodology

The visualiser is an extensive new addition to the existing System Designer application; while this software addition and the consequent changes required detailed technical checking they also required significant pedagogical checking as well. This chapter begins with a discussion on methodology relating to this research in section 7.1 while section 7.2 presents operational constraints on the study. The research questions are developed in section 7.3 and the research methods selected to answer these questions are in section 7.4. Section 7.5 discusses the methods for exploring the first research question and section 7.6 those for the second research question. The justification for the choice of participants is discussed in section 7.7, and ethical considerations are covered in section 7.8. The research tasks developed to support the research are presented and discussed in section 7.9.

7.1. Choosing a methodological direction

Methodology and research questions are in a two-way relationship, as research questions determine the choice of methodology and the choice of methodology will constrain research questions (Case & Light, 2011). In this research an existing tool has been significantly extended and altered. The learning benefits of these changes are as yet unproven, so a qualitative or inductive methodology was chosen to test them as it provides "elaborate accounts of human perceptions" (Leydens, Moskal, & Pavelich, 2004, p. 65) and fuller perspectives on the meaning that learners will be developing (Kaplan & Maxwell, 2005; Myers, 1997). An initial general study would also highlight as many deficiencies in the tool as possible as well as provide the needed insight into the limitations of the tool as an analogy of a real system (Gilbert, 2004). It consequently should provide indications of changes needed before implementation into the full classroom program and further specific research goals.

It is the understanding of learners' mental structures or models which are central to this research. Understandings of these with regard to learning to program have been investigated through a number of phenomenographical studies (Ben-Ari, Berglund, Booth, & Holmboe, 2004; Booth, 1997, 2001; Richardson, 1999; Thuné & Eckerdal, 2009). A phenomenographic approach is concerned with categorising the variations in the views of users; one significant recent study has used the phenomenographic approach to categorise program understandings (Thuné & Eckerdal, 2009) these were used to categorise students understandings in this research. While research into

student understandings of embedded systems is not as developed it is an area well categorised by research literature, as discussed in section 2.1, and an analysis using these categories was seen as most appropriate.

7.2. Operational constraints

Testing of a pedagogical software application is non-trivial because of the wide range of highly complex and co-dependent human and contextual variables involved in learning. This could lead to a wide range of interesting options for testing; however what a researcher wants to know does not always align with what can realistically be investigated (Fincher & Petre, 2004). A number of constraints were consequently identified and guided this research.

The primary focus of this research was to build a useful tool for both students and educators of embedded systems; thus the position was taken that both were equally important as participants in the testing of the visualiser. This reflects the importance of teachers' roles in learning as they are responsible for developing and managing learning environments; as well as this they hold the key to what takes place in their classrooms so gaining their buy-in to using the tool is essential. The importance of students in testing is to see if the software supports the development of viable mental models (or otherwise); this reflects the student being at the centre of the learning process i.e. both cognitive and constructivist views of learning. Accessing both groups of participants was seen as vital, however it also meant limiting the number of participants so that results could be analysed as deeply as possible.

One constraint related to the phenomenographic data analysis, a full analysis would require time that this study did not have. It would include using the visualiser over a number of cycles and with a significant number of participants (16-20 is recommended), followed by an iterative approach to categorisation of data by at least two researchers to avoid researcher effects (Kinnunen & Simon, 2012). As this approach was not feasible in the time available it was determined that this research would include only a pilot study that provided feedback to inform the final stages of development prior to introduction into the learning program.

A crucial operational decision concerning research into the visualiser involved the choice of context and participants. The researcher was on study leave from a secondary school teaching position and while secondary school students and teachers could have been used for the testing, access to a tertiary environment with older students and experienced lecturers was seen as an opportunity that would not be as freely available in the future. This was not seen then as a

limitation that might affect generalisability of results but as a way to broaden the contexts in which the tool could be useful in the future.

7.3. Research questions

There are two aspects to a computer application used for learning purposes: its usability and its efficacy. A pure usability study however leads to a shallow understanding of a tool (Urquiza-Fuentes & Velázquez-Iturbide, 2009) so the usability question was broadened to a usability analysis of the tool within the bounds of it as a learning environment:

- RQ1. How successful is the visualiser as a model based learning environment?

An educational tool (however good it looks) is still guided by the assumptions made by its designer while developing it (Stern, Markham, & Hanewald, 2005). A tool must be judged against its fitness for purpose and in the absence of some analysis of its efficacy leads to the stereotype that "there are only two types of tools: tools people don't use and tools people complain about!" (A. Malik, personal communication, 17 October 2013). While research and existing visualisation trials have justified development of the visualiser it is important to gain some specific details about its efficacy and the second research question is:

- RQ2. In what ways does the visualiser benefit novice learners of embedded systems?

7.4. Selection of research methods

The last decisions were the choice of methods used to investigate the research questions. The categories chosen to analyse data were taken from the phenomenographical analysis of learning to program (Thuné & Eckerdal, 2009) and to provide a measure of reliability to these results the same qualitative methods were chosen for this study. Qualitative research methods used in phenomenographic analysis include interviews, observation and analysis of outcomes (Boustedt, 2008; Ebenezer & Fraser, 2001; Marton, 1981). While interviews are useful to gain explicit comments, observations of users are, however, more powerful since they deliver more insight into the implicit mental structures of participants.

7.5. RQ1. Success as a model based learning environment

The characteristics of good models (Mayer, 1989) were used to guide and shape the development of the visualiser so these were used to guide the investigation. To increase trustworthiness of any results, triangulation of data is an important consideration (Leydens et al., 2004) so different

methods and participants were used. This included discussion with lecturers, observations of students and a questionnaire given to students (Appendix E: Visualiser experience questionnaire). The questionnaire used a Likert scale, a recognised method of gaining participants' responses to a phenomenon (Carifio & Perla, 2007). The six items asked are in Table 7-1. Participants mark their agreement or disagreement with each statement on a scale (see questionnaire in Appendix D: Programming and Embedded Systems Experience Questionnaire) and were asked to provide verbal feedback as well.

Characteristic	Agreement statement for novice learners questionnaire
Complete	Visualiser contained the needed information to do the tasks (nothing was missing)
Concise	The Visualiser contained just the right amount of information (not too complicated)
Coherent	The visualiser made the way the software and hardware worked together make sense
Concrete	At the end of the tasks the visualiser seemed natural and familiar to use
Conceptual	The visualiser made programming more meaningful for me
Considerate	Visualiser words, images and diagrams organised in a student friendly way

Table 7-1: Likert item statements used to gain feedback on model characteristics

The seventh characteristic of model based learning relates to the correctness of the representations in the visualiser. Correctness was tested against two reference texts (Barnett, O'Cull, & Cox, 2007; Kelly & Pohl, 1990) and a commercial C-interpreter (Soft Integration, 2013), by demonstration to lecturers and inferred from students' use of the visualiser.

7.6. RQ2. Benefits for novice learners

Two separate aspects of student understanding are of interest when investigating results. The first is how students perceive the unique learning aspects of embedded systems such as hardware-software interaction and its reactive nature. The second aspect is how students see program code and what benefits the visualiser might bring in learning to program. A range of ways of understanding programming exists as discussed in sections 2.3.4 through 2.3.7 and provided insight when analysing the results.

To gain an understanding of how users experience the visualiser both an explicit and an implicit investigation of participants' experiences was undertaken. Educators with their understandings of embedded systems and programming, as well as their experienced views on teaching and learning, were able to provide explicit reactions to the visualiser. These were captured using semi-structured interviews. This form of interview uses open-ended questions that initiate discussions and thus allowed the lecturers an opportunity to reveal their own experience of the visualiser without the constraints of predetermined categories.

While educators have qualified opinions to offer, they are experts and tend to forget the difficulties associated in learning even simple concepts. To gain valid feedback from learners required a shift toward gaining an implicit reaction to the visualiser as well as an explicit one. The students were encouraged to offer explicit comments about the visualiser and the tasks. However, by having students carry out the tasks using the visualiser (and observing them while they do so) they provided an implicit view of their mental models of embedded systems and any changes to their model that might have come about from use of the visualiser. This inference of mental model development via tasks is a recognised method of data collection for novice learners (Jonassen, 1995; Perkins, 1991).

A highly significant factor in research of the visualiser with learners is that while their experiences can be observed and analysed from the way they use the visualiser, they are doing so in the context of the tasks created for them. This means then that any effects identified are not related solely to the visualiser but are combined with the effects of the task as well. This aligns with what has been discussed in section 6.1.2 that quality learning only comes from engagement with visualisers during the undertaking of meaningful tasks.

7.7. Participants

Participants for a study are often randomised to provide statistical representation while other known variables are highly controlled In this qualitative research the variation in participants was strategically controlled to provide as broad an understanding of the unknown variables as possible without excessive student variation confounding any results. When testing modelling environments participants are controlled based upon their understandings of domain knowledge and modelling knowledge as shown in Figure 7.1 (Gemino & Wand, 2004).

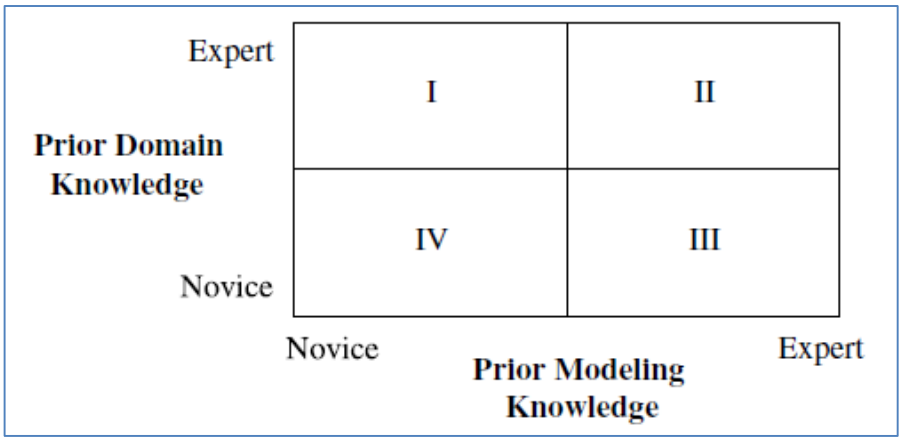


Figure 7.1: Participant selection criteria (Gemino & Ward, 2004)

When selecting participants for this study it was important to select participants based upon three dimensions which included modelling and the two knowledge domains of embedded systems and programming. This led to a three way selection process as depicted in Figure 7.2.

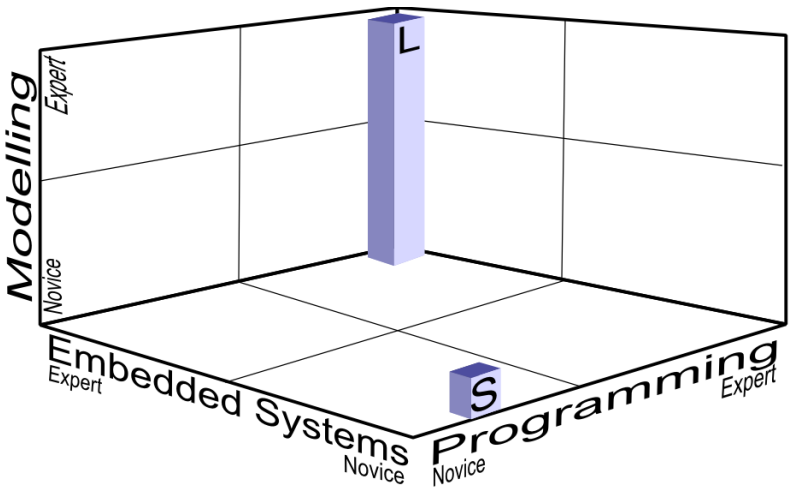


Figure 7.2: Student and lecturer participants experience selection criteria

The student participants selected for the study were novice students in a stage 2 computer systems course in the Faculty of Engineering. These students had already undertaken a half semester course in MATLAB and a half semester course of programming in C and were currently

learning about C++. They were selected from within the course as they had no previous experience with embedded systems, they had no other programming experience outside their courses and they had no experience with program visualisation. This group of students were preferred over complete programming novices, as it was felt that students completely new to both programming and embedded systems would struggle with programming, and not be able to contribute to the understandings of the visualiser as a tool for building mental models of embedded systems. It was also felt that students who had a great deal of programming experience but limited embedded systems experience would bring too many confounding factors into the testing environment, particularly in relation to their experience with programming environments and visualisation or debugging tools.

While tertiary students who have had some programming experience were preferred for the study, they are a quite different group to the secondary students that the tool is initially being developed for. This relates specifically to their academic levels of reasoning and thinking. To reflect these different levels some of the tasks were made more challenging than those that would be introduced to 15 year old students in the classroom. These changes were drawn from understandings of differentiation of instruction. Differentiation is a framework for thinking about the alignment of the processes, content or outcomes of instruction with learners' abilities (Tomlinson & Allan, 2006; Tomlinson, 2008; Tomlinson et al., 2003).

Testing of the visualiser with students was carried out with each one individually and took approximately 90 minutes. This involved a short programming experience questionnaire, undertaking of a range of tasks using the visualiser and a summary questionnaire and discussion.

University lecturers in embedded systems were chosen as a further group who had valid experience of all the dimensions of interest. While lecturers are not representative of the group of electronics teachers at secondary school level they have expertise in all three dimensions of Figure 7.2. In general secondary school electronics teachers have limited experience in electronics, embedded systems and programming as they have often come from either a materials technology or science background and developed a passion and range of understandings for the area over time. Testing with university lecturers will not only leverage off their expert understandings but they also represent a potential target group for use of the visualiser as well.

7.8. Ethical considerations

Ethics approval for the research project was granted by the University of Auckland Human Participants Ethics Committee (Reference 2013/10032 on 30 July 2013).

7.8.1. Conflict of interest

While the researcher is on funded study leave from the Ministry of Education, there is no potential conflict of interest as there is neither involvement by the Ministry of Education in the research nor any requirement to provide or publish results from the study.

7.8.2. Confidentiality and consent

All participants were assured of the confidentiality of the data collected. All student participants' data was collected in a laboratory separate to the work areas the student participants use, so as to provide them with confidentiality. Student participants were offered the chance to withdraw or review any data up until two weeks after the data was collected, and they were given an assurance that their participation and any results would not have any effect on their course grades.

7.8.3. Exclusion of potential participants

All participants were initially screened via email with regard to their experience with embedded systems and programming so that only novices were asked to participate. There were seven students in the course who were already familiar with embedded systems as they had taken electronics at Mt Roskill Grammar School, and they were therefore excluded from participation.

7.9. Task Development for pilot testing

In Chapter 6 the exercises for use in the classroom have been discussed. These however build on a foundation of practical tasks with actual hardware where students have already made LED sequence circuits. The participants in this study are not from that setting so a set of introductory exercises were developed to train the participants on using the visualiser. These tasks closely resemble those carried out with actual hardware in the classroom.

7.9.1. Participant task A

Task A is shown in Figure 7.3 and serves as an introduction to embedded systems and the visualiser. Students can run the visualisation but are not required to make any changes to the hardware or software. Participants are instructed on how to open and close the software, run, pause and step the code. The interaction between each line of code and the hardware (the output LED) is not explained to participants but left for them to infer from single stepping the program (Figure 7.4). Questions often used in the classroom to build this understanding are: tell me about what 0b10000000 means? If the LED was on A.0, what would the line of code become?

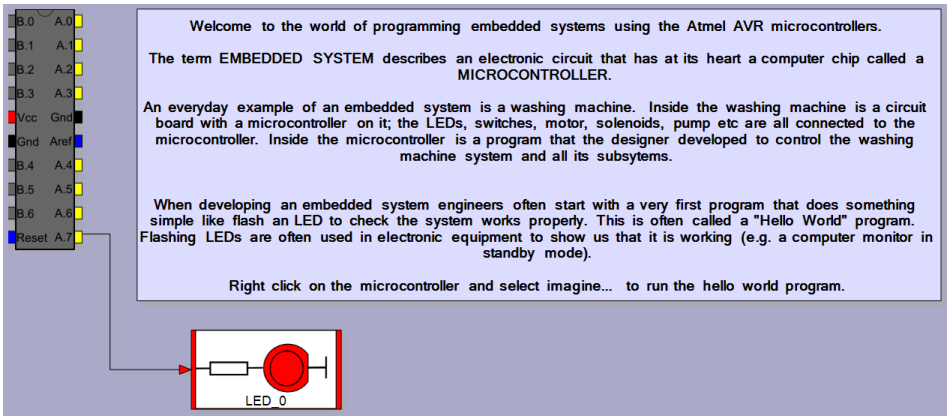


Figure 7.3: Task A, Hello World!

```
2  //to run this program you will need an LED connected to PORTA.7
3
4  DDRA = 0b10000000;          // configure PORTA.7 as an output
5  while (1)                   //loop forever (while (1) is always true)
6  {
7      PORTA = 0b10000000;     //LED A7 on
8      _delay_ms (500);        //wait 1/2 a second
9      PORTA = 0b00000000;     //LED A7 off
10     _delay_ms(500);         //wait 1/2 a second
11 }                           //go back to the while
12
```

Figure 7.4: Program code for 'Hello World' task

7.9.2. Participant task B - Knightrider

Task B in Figure 7.5 relates to the hardware exercise specifically developed in class over time so that students would have to actively engage with the different aspects of context, syntax and sequence of program code. Syntax understanding builds on the understandings from the first task (the number of bits in a byte, how each bit controls one specific pin of the microcontroller and the order of the bits in a byte). Sequential processing of code is introduced in the exercise as the LED pattern goes in both directions, so new code must be inserted into the centre of the sequence. Contextual understanding is targeted by the non-repetition in the sequence of the first and last LEDs. The sequence is A0-A1-A2-A3-A4-A3-A2-A1 as per the code in Figure 7.6. A0 appears only once in the sequence just as A4 appears only once.

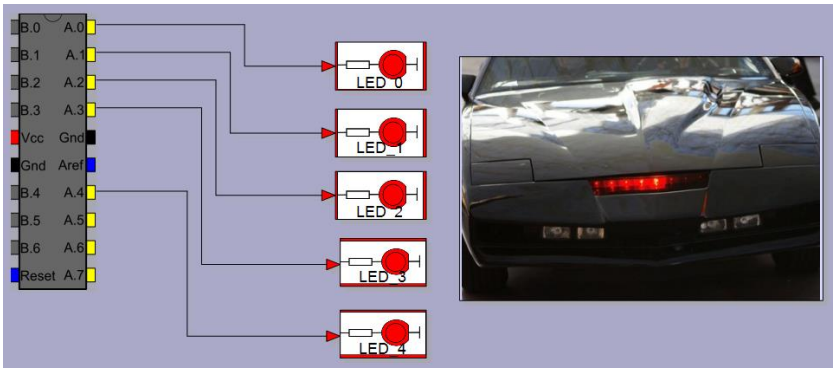


Figure 7.5: LED sequence task B and its contextualised environment

```
1  //can you add another 3 leds to the sequence for the knightrider car
2
3  DDRA=0b11111111; //make all the pins on PORTA outputs
4
5  while (1) // repeat forever
6  {
7      PORTA = 0b00000001;  //A.0
8      _delay_ms (500);
9      PORTA = 0b00000010;  //A.1
10     _delay_ms (500);
11     PORTA = 0b00000100;  //A.2
12     _delay_ms (500);
13     PORTA = 0b00001000;  //A.3
14     _delay_ms (500);
15     PORTA = 0b00010000;  //A.4
16     _delay_ms (500);
17     PORTA = 0b00001000;  //A.3
18     _delay_ms (500);
19     PORTA = 0b00000100;  //A.2
20     _delay_ms (500);
21     PORTA = 0b00000010;  //A.1
22     _delay_ms (500);
23 }
```

Figure 7.6: LED sequence task B starter code

7.9.3. Participant task C – EMDR light bar

Task C (Figure 7.7) involves a longer and more complex sequence with the associated problem to solve as the sequence crosses the port boundaries as discussed in section 6.2.1 on page 74.

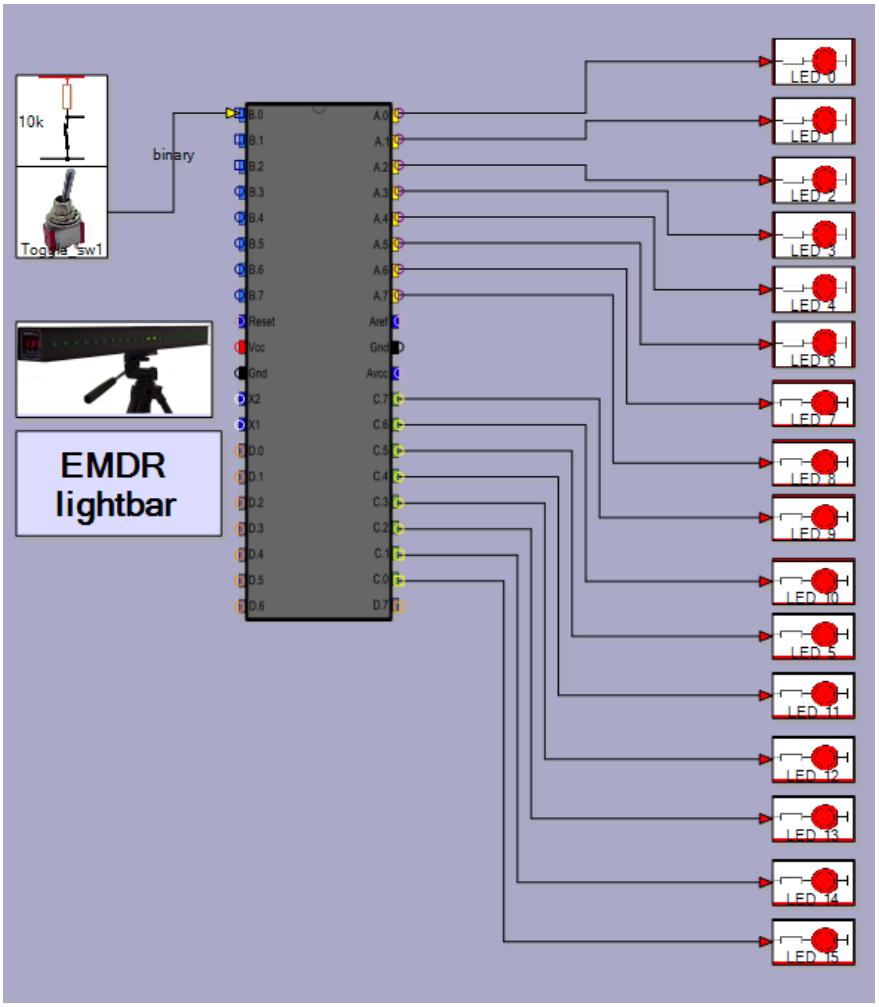


Figure 7.7: EMDR LED sequence task C showing its contextualised situation

A number of errors were purposefully introduced into the program code (Figure 7.8) to see how students used the visualiser to identify errors typical of those that commonly occur in real hardware in the classroom. These were all introduced in a single task; when these are encountered all together in one instance in the classroom most first year students would struggle to identify and fix all them all. These were included in one task to leverage off the higher level of reasoning and logical thinking capability of second year university novice embedded systems learners, compared to 15 year old secondary school novice embedded systems learners.

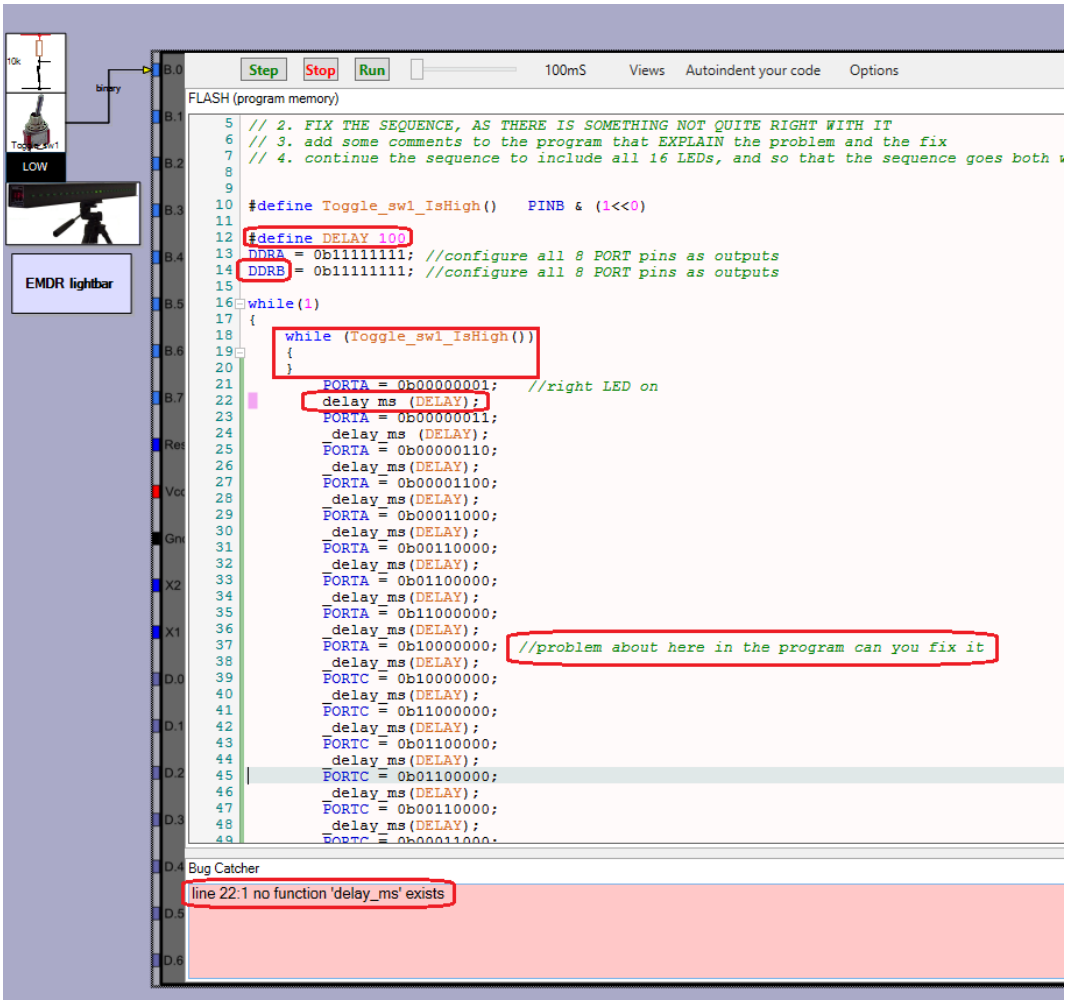


Figure 7.8: Sequence task C, showing errors and new features

The introduced errors and new programming features are summarised here along with the reasoning behind them:

- Feature: the new and larger ATmega16A microcontroller; in the classroom first year students practical work is limited to the ATtiny461. The visualiser allows the abstraction of conceptual understanding about microcontroller ports by introducing another microcontroller that is similar but different.
- Feature: line 12 – the use of #define has been introduced to make the delay easier for the programmer to change; this also features in the classroom exercises.
- Error: line 14 – DDRB is configured instead of DDRC. In AVR microcontrollers the Data Direction Registers (DDRA, DDRB, etc.) are initially configured as inputs, so the port will not work at all. Forgetting to configure a port or pin as an output is a common error encountered with students' programs, and is typically a very challenging bug for novice learners.

- Feature: lines 18-20 – the 'while' loop was introduced with a toggle switch to control it. This does not normally feature in the classroom so early in learning, but was added to the participants' exercise as they already have programming experience.
- Error: line 22 an error was introduced; *delay_ms(),* rather than the correct *_delay_ms(),* to identify participants' use of the error window.
- Feature: the sequence was changed from a single LED moving pattern as per the last exercise to a double LED moving pattern (as used in an actual EMDR light bar). In the classroom students are given the task of a single LED pattern and then progress to the double LED sequence. This was changed for the participants in the pilot study to reflect their more capable thinking and reasoning levels.
- Feature: some guidance has been given in the program as to where the error in the sequence is to be found (comment on line 37 in the program code). This is different to the usual classroom exercise where students are given no support. This was added to the participant exercise to reduce the time involved for the task, but not its complexity.

7.9.4. Participant task D – controlled pedestrian crossing

This task is the same task as in section 6.2.1 and Figure 6.2. No changes have been made to it to differentiate it for the more advanced level of novice university student participants.

7.9.5. Participant task E – VOR Morse code

This task is the same task as in Figure 6.3. No changes have been made to it to differentiate it for the more advanced level of novice university student participants.

7.9.6. Questioning of learners

In class, questions are used after practical exercises to make sure students take away from the exercise the important learning points. Questions were not used with participants as it was the learning interaction with the visualiser that was being investigated.

7.10. Summary

This chapter has outlined the aims that guided pilot testing of the new visualiser within System Designer and the supporting methodological and operational decisions that were made. Phenomenographic research methods were identified as highly suitable for gaining an appreciation of the experiences of learners while using the visualiser. The two research questions

were discussed and the specific methods used to investigate each were explained. The participant selection process was discussed and justified in terms of the results that these specific participants could contribute to understandings of the visualiser. Ethical research practices are a crucial part of any investigation using human participants and the details relating to this were presented. The chapter also gave details of several of the exercises used with learners and the reasoning behind how they differ from those used previously within the school context. In this chapter the goals of testing were explained that it was to provide: as much pertinent data as possible within the available timeframe, insights into constraints associated with the visualiser, identification of visualiser bugs, guidance for the visualiser's final development before implementation into a programme of learning and also guidance for further research.

Chapter 8. Results

This chapter draws out the results from pilot testing of the visualiser with students and the demonstrations and discussions with lecturers. This chapter begins by reviewing the study participants in section 8.1 and data collection methods in section 8.2. Coding of data is explained in section 8.3. Results relating to the first research question are identified in section 8.4 and for the second question in 8.5. Section 8.6 presents significant results that indicate changes that should be made to the visualiser, to the tasks and to System Designer. The data (observations and comments) captured during the tasks, discussions and demonstrations are coded in Appendix G: data coding

8.1. Participants

The student participants were four second year engineering students who had no background in embedded systems but had two single semester courses in programming (MATLAB and C). Each student was tested individually, however student A and student D were friends who generally worked together in a group with a few others; also student B and student C were friends who always worked together. They were asked to not discuss the experiment with their friends until after they had each completed it. The four lecturers were all experienced university lecturers; three had strengths in embedded systems and one was a computer science lecturer.

8.2. Data collection methods

Student participants answered a questionnaire to identify their programming and embedded systems experience. They undertook tasks during a 90 minute session using the visualiser and their actions were captured using screen capture software. Observations were extracted by reviewing the screen recordings. Notes were taken of comments they made and these were collated with the screen recordings. A second questionnaire was used to capture their overall reactions to the visualiser.

The lecturers had a short demonstration of the visualiser, and a set of semi-structured interview questions were created to guide a discussion. All four lecturers however generated enough discussion about the visualiser without the need to use most of the questions directly.

8.3. Coding of data

Coded data is collated in Appendix G: data coding

The student data has had colour coded comments attached to it to identify how it relates to each research question. The data coding was carried out by reviewing each screen recording at least three times to identify actions that indicated the level of student understanding. The actions used to identify codes were: student focus on text without appreciation of its action; student focus on getting the program to work as opposed to a line of code to work; time taken to carry out a task or one aspect of a task; time between reading an instruction and correctly carrying out some action; amount of movement through code searching for something; incorrect or trial and error actions; aimless changes to program code; time reading instructions; whether reading of instructions or carrying out actions came first; amount of re-reading of instructions; any 'aha' comments; correctness of terminology used; questions asked; any assistance requests and verbal comments made.

8.4. RQ1: How successful is the visualiser as a model based learning environment?

Data for the success of the visualiser as a learning environment related to two separate aspects: the first was how well the visualiser performed against the seven characteristics of good models and the second was to identify its limitations as an analogy or model based learning environment.

8.4.1. Completeness - the visualiser contained the needed information to do the tasks (nothing was missing)

In the questionnaire three of the four student participants gave complete agreement with this statement. Student C commented that he had to ask for some assistance at the beginning. A comment from student A provided confirming evidence for this characteristic. While carrying out the EMDR light bar task he said, "where is the LED connected? Oh I can see it there", signifying he was able to solve the problem for himself with the information available in the visualiser and supporting the visualiser's characteristic of completeness.

The completeness characteristic is more fully supported by lack of contradictory evidence as the participants worked through each task directly without the visualisation environment appearing to hinder their progress in any way. A significant result concerning lack of completeness however was common to all four student participants; this was that the error window (bug catcher) did not

satisfactorily attract their attention. This is an example of actions that students did not do, providing contradictory evidence. For all participants the window was visible on the screen and errors were displayed in it as they occurred during parsing of the code, however students did not respond to the error messages. After the first two trials with participants the error window was changed so that when an error was encountered the background changed to red; again however during the next two trials the students still did not attend to the error window messages.

8.4.2. Concise - a summary only capped at five or so common tasks to avoid excess information

One of the final comments made by a student was "it's so simple to see everything operating". This indicates the student's ability to unpack what he was seeing and not be confounded by extraneous information. Whilst this comment supports conciseness, this characteristic is more strongly supported by a lack of contradictory evidence. Where students quickly identified what they needed to do and carried it out without resorting to a trial and error approach, or searching for something then it would strongly suggest the visualiser as being concise. This was the case for all four students as they carried out the tasks.

The most demanding aspect of a task presented to the students was in the EMDR light bar task where the program code contained an error for the configuration of the output port (DDRB was used in the configuration instead of DDRC). This was more demanding than any other task aspect as the error appears to occur in one part of the program (output code) however it must be resolved in another part of the program (the configuration section). Three of the students quickly realised that their output code was correct and they needed to review other parts of the program; they then only took a short period of time to identify the error. Two of the students changed it without question and one of them asked if he should. Participant C did not look outside the output code area of the program code and had to be instructed to change the line of code after clearly struggling with the error. This was one indication of the different level of programming ability between participant C and the other students. For three of the participants the visualiser presented a concise enough environment as it allowed them to find an unfamiliar error; however for a weaker student it was not concise enough to find the difficult error.

8.4.3. Coherent – sense making, showing interactions and rules for those interactions

One student and one lecturer both made positive comments concerning the colours (see Figure 8.1) used to distinguish between true and false in Boolean tests, with both saying they were "very useful".

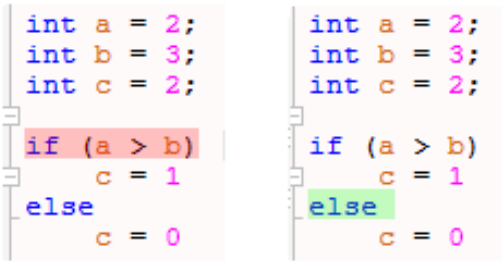


Figure 8.1: Boolean test colouring

Another student said during a task "it's not turning off when…." and then immediately saw what he needed to change in his code to make the LED change. These comments indicate that the students did not struggle to understand what was happening when the programs were running.

The visualisation of embedded systems program code extends further than understanding of programming but also encompasses the full process of embedded systems development; this process was commented on by student C as being made coherent:

> "it's really useful because working with other programs you have to compile, recompile and then run it and then check what's wrong with it whereas with this, when I'm stepping through it I can see like the output here as well whereas in there it's like harder to see what you've mistaken".

8.4.4. Concrete – depiction should be familiar to the learner

The visualiser was envisaged as complementing a course in embedded systems which has a predominant practical and experimentation focus so was developed to mimic real hardware as much as possible. While the students who participated in the study had no experience of microcontrollers (but had prior experience with electronic circuits) their understanding of the concreteness of the visualiser as it relates to prior experience is limited. However one comment a student made clearly identified concrete depiction of the visualiser; "it's easy to see how the input and output signals go between the components and the microcontroller" indicating the switch input and LED output circuits and their changing voltage levels indicated by the link colours

(Figure 8.2 and Figure 8.3). Lecturer C commented "the voltage divider on the switch is very nice, helpful for students to see the voltage levels change and link the switch to input code".

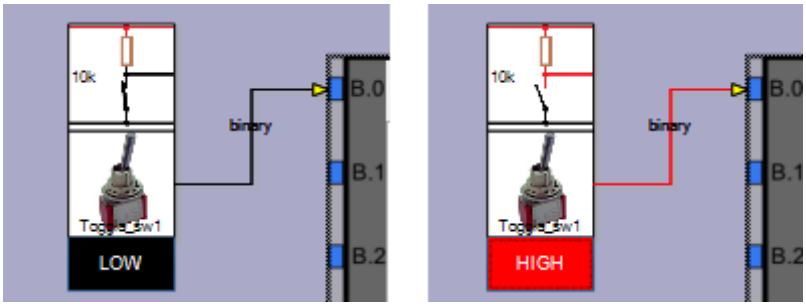


Figure 8.2: Visualiser showing low and high states of the switch input circuit

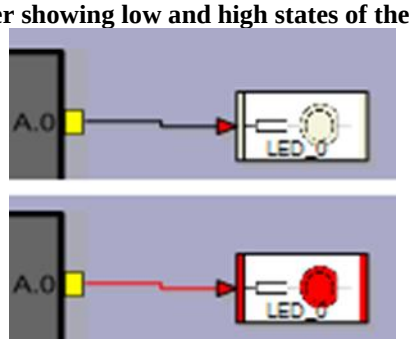


Figure 8.3: Visualiser showing low and high states of an LED output

8.4.5. Considerate - student level vocabulary and organisation

All of the students indicated the visualiser was easy to use, and none of them showed confusion at any stage with the visualisations. Two of the lecturers questioned the choice of BASIC as a learner's language rather than only using C; another lecturer however questioned the use of C as a language choice for school students as it is a difficult language for new users. Evidence of consideration came from lecturer A who said the visualiser "is simple to use, something that students can latch into immediately". Evidence of consideration also came from the lack of questions from participants asking what aspects of the environment were or meant.

8.4.6. Conceptual – shows the system operation meaningfully

The students all indicated agreement for this characteristic and made positive comments about understanding. In terms of the environment of the visualiser itself the comment made by student C after the back yard alarm task was "I can see like the output here as well whereas in them [a real system] it's like harder to see what you've mistaken". The students comment compares the visualiser with the usual flow of programming, his comment directly relates to his understanding and how the visualiser made it easier for him.

Significantly the lack of struggle by students overall to complete a range of exercises involving a technology they had no experience with is supporting evidence for the visualiser's influence on assisting with conceptual understanding. A further indication of the visualiser's positive influence on conceptual understanding came from the comments about colour coding of the Boolean tests (as described in section 8.4.3); these were directly described as supporting understanding by one lecturer and one student.

8.4.7. Correct - are the major analogies used correct?

Overall the visualiser was viewed by lecturers as correctly implementing an embedded system; this was evidenced by their comments referring to extra features that could be implemented and also to how it might support learning for their students rather than correcting errors within its operation or interface. One comment included how the visualiser could be used to support students in embedded systems during the first three years of their learning at undergraduate level, and how it could be of benefit to even post graduate students with extensive programming experience but no embedded systems knowledge.

A number of minor correctness issues were observed during testing; one important consideration was that when running program code in the visualiser that did not have a while(1){} loop the program stopped execution at the end of the program code. This is not the situation in a real microcontroller, as with real hardware a program that lacks a while(1){} resets after reaching the end of the program and restarts the program again. This has been noted on a number of occasions in class with students' programs and leads to unintuitive effects in program operation, effects which novices do not easily interpret. After student testing it was decided to implement the correct nature of an embedded program, namely one that resets rather than stops so that the visualiser more closely represents actual program action.

8.4.7.1. The 'two-delays' issue

Testing with students did reveal a significant limitation with correctness of the visualiser as an analogy of an embedded system. In introductory programs fixed time delays are often introduced to pause the microcontroller so that a human can observe system behaviour, i.e. the use of _delay_ms within the EMDR program task (Figure 7.7). When using the visualiser the effect of a _delay_ms command in the program code is complicated by the delays introduced when single stepping through the program code or running the program code at slow speed.

This is explained using Figure 8.5. The five image captures (A, B1, B2, C1 and C2) are those seen as the program code is stepped through line by line. The captures of B1 and C1 however are never seen when a program is run at full speed because they have no delay commands associated with them. Consequently the mind perceives the sequence correctly as in the images A, B2 and C2.

Users need to conceptually separate these two delays, with the first relating to the program code and the second relating to the visualisation environment. Two students had no difficulty with this and separated the delays in their mind completing the task with little difficulty. One student realised the issue related to the visualisation and developed his own strategy to resolve the confounding effect of the two delays by restructuring the code so that commands were on the same line of the program as in Figure 8.4 and thus appeared to execute simultaneously when single stepped in the visualiser. One student struggled a great deal with the issue and did not resolve it.

```
PORTA = 0b01100000;
_delay_ms(DELAY);
PORTA = 0b11000000;
_delay_ms(DELAY);
PORTA = 0b10000000;  PORTC = 0b10000000;
_delay_ms(DELAY);
PORTA = 0b00000000;  PORTC = 0b11000000;
_delay_ms(DELAY);
PORTC = 0b01100000;
_delay_ms(DELAY);
PORTC = 0b01100000;
 delay_ms(DELAY);
```

Figure 8.4: The two delays issue solved by one participant

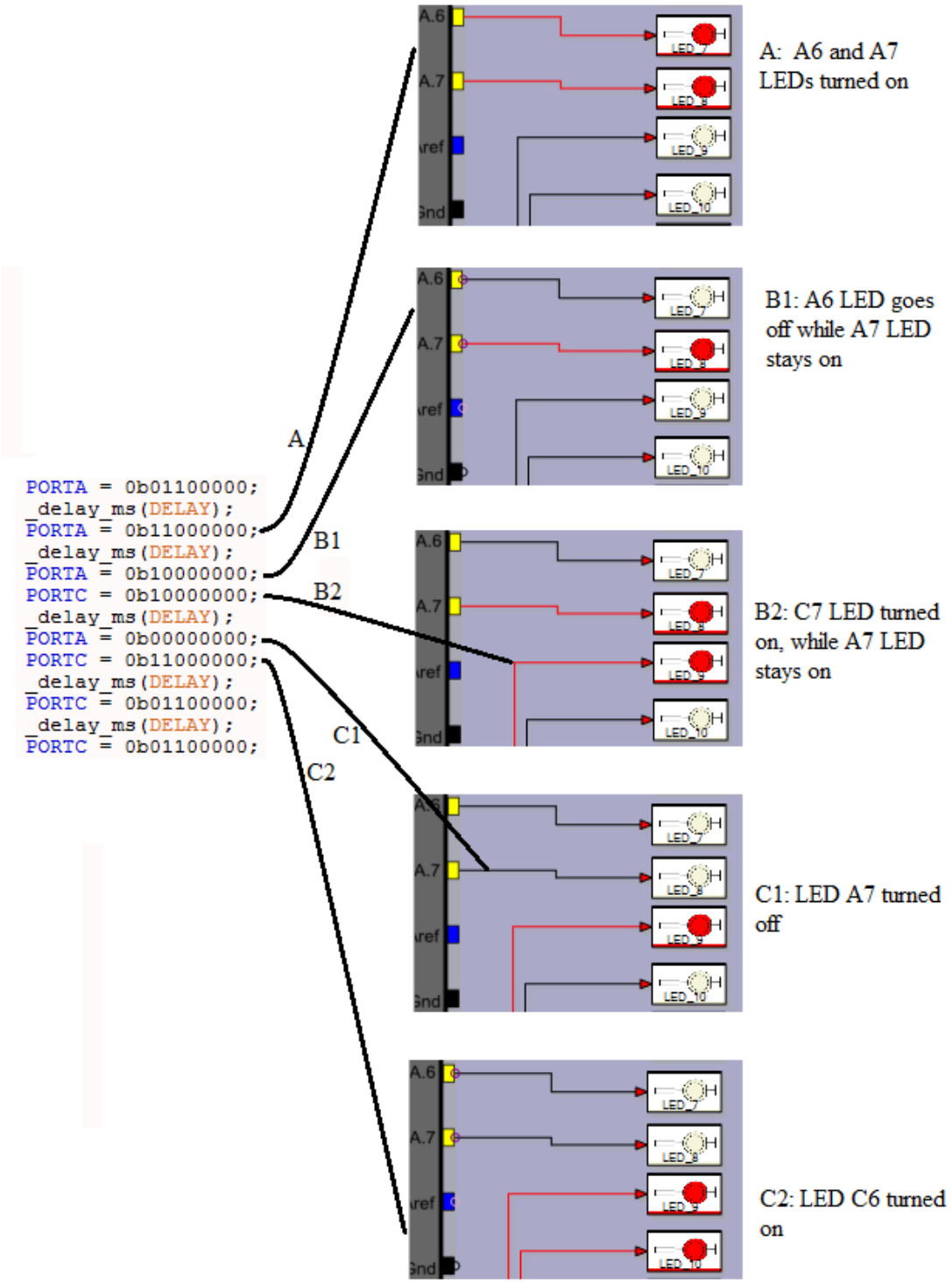


Figure 8.5: EMDR sequence at port changeover

8.4.8. Constraints for the visualiser as an analogy of an embedded system

An important result from testing the visualiser is the identification of the constraints under which the visualiser can be reliably used with learners as an analogy or model for an embedded system. As noted in section 8.4.7.1 when the visualiser is not run at full speed (i.e. it is either in single step mode or running code with a fixed delay) then timing aspects of the program are not correct

and are difficult for novices to interpret. This presented a great deal of confusion for student C, to the point where he did not fully complete the task but thought that he had.

8.5. RQ2: In what ways does the visualiser benefit novice learners of embedded systems?

In this section the outcomes of testing with students and lecturers are identified in terms of how students might benefit from using the visualiser. Of interest are the two knowledge domains of embedded systems and programming. For embedded systems this means that the student begins to show a growing awareness of aspects such as hardware-software interaction, the reactive nature of embedded systems, synchrony and criticality as discussed in section 2.1. In Appendix G observations and comments that showed student awareness of hardware-software interaction are labelled ES-HSI, reactivity in embedded systems is coded as ES-R and the critical nature of embedded systems is coded ES-Critical. The other aspects of embedded systems (concurrency and synchrony) did not feature in the results.

8.5.1. Hardware-software interaction

Data revealed how quickly the students appreciated the interaction of hardware and software in an embedded system. At the explicit level all four of the students contributed comments that supported their developing understandings of the interrelatedness of hardware and software: “it helped because I didn’t know how C handled working microchips, making it actually physically do things", “it’s easy to see what happens because of the LEDs and switches”, “when I’m stepping through it [the code] I can see like the output here as well” and “so simple to see everything happening”.

While observing students, two different interrelationships between hardware and software were noted. A direct relationship was indicated where students grasped which I/O directly related to which line and command of program code. This was evident throughout the testing in that all students, in all tasks, quickly and without resorting to trial and error, identified what they needed to do to control some I/O. One example was the common error made by novices with binary notation. Programmers often incorrectly enter byte-sized binary numbers with seven or nine bits rather than the correct number of eight. Two of the students made this error in the EMDR light bar task; where the only indication of the error was the incorrect sequence of the LEDS. Both students quickly extrapolated the error as relating to the incorrect number of bits and changed the values without resorting to changing any other program code.

Students' understanding of an indirect relationship between hardware and software was observed in the EMDR light bar task as it had an incorrect data direction register configuration. This required students to see that the non-working LEDs related not to the line of program code that turned them on and off, but to a completely different line of the program code that controlled the port the LEDs were connected to. Three of the students identified the error quite quickly; upon identifying a problem with the LED sequence they rechecked the code that directly related to the LEDs then looked through the code and quickly identified the incorrect line. Two of the students immediately changed it and the third asked whether he should. Student C however did not look beyond the commands that directly related to the LEDs and had to be guided to change the error.

The students' developing understanding was reflected in their quick grasp of microcontroller hardware terminology, using terms such as port, output and input rather than gestures or the word 'thing'.

8.5.2. The reactive nature of embedded systems

Continual reaction or response to the environment is the second key understanding of embedded systems. Three separate results were identified. The first was referred to as the two-delay issue, where the delays during single or slow stepping of program code and the delays due to _wait_ms() delay commands in program code confused one student. This was a known issue and highlighted as a limitation on correctness for embedded system visualisation in the analysis of visualisers in section 5.3. The second was that different switch types (e.g. push button and tact switches) needed different visualisation characteristics during single or slow stepping of program code. The third was that program code without an enclosing while(1){} did not restart but stopped executing.

These could lead to misunderstandings for novices of the reactive nature of embedded systems. The first must be resolved in the way the visualiser is explained to students so that the two delays are clearly and explicitly separated for learners. The way that student B resolved this for himself by putting the two delay commands onto a single line (see Figure 8.4) would be a useful strategy for this.

The second has been covered by changes to the visualiser model for switches. The push button and tact switch models have been changed to have two modes of operation. During run mode of the visualiser they act normally as push-to-make and release-to-break contacts. During single-step mode of the visualiser they have an automatic time out so that once the switch is pushed by

the user it automatically breaks in the circuit after five seconds, thus allowing time for the user to single step through the code that detects the switch press. This would also enhance their understanding of reactivity, as if program code does not detect the switch action in time it will miss the switch press, just as in a real embedded systems that is polling switches.

The third issue of reactivity was encountered when a student removed the while(1){} in his program and the program halted (see Figure 8.6). This has been changed so that the program restarts from the beginning again if the end of code is reached. Future testing could explicitly explore this reactivity error with students.

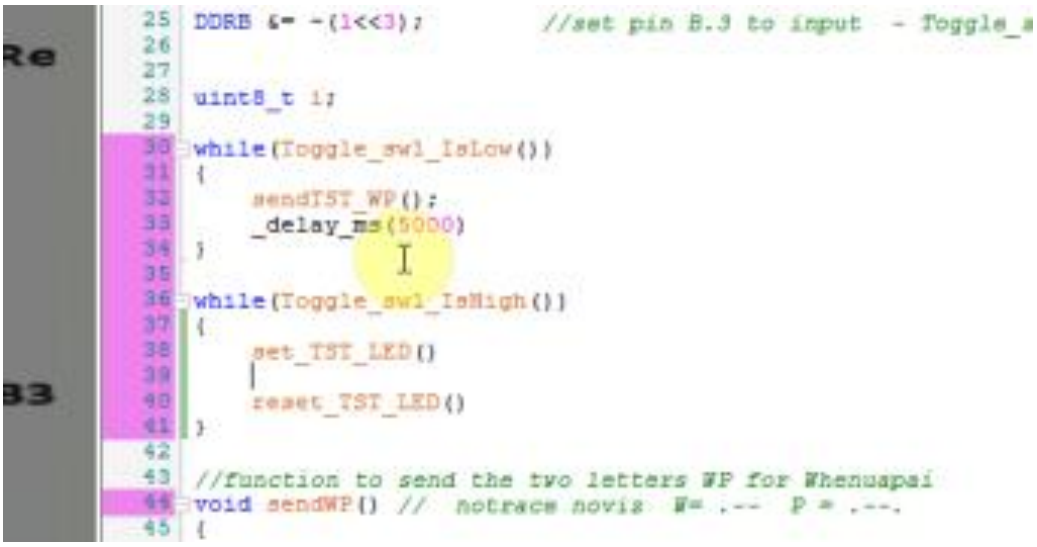


Figure 8.6: Student removes the while(1) code pattern

8.5.3. State in embedded systems

Understanding of state is a feature of several of the tasks. In the EMDR light bar task three of the students successfully completed the exercise correctly however student C did not notice that an error still existed in his final code (which included an extra delay). In the backyard alarm and VOR Morse code tasks the concept of state is present with the override switch being used to set the state of either the alarm or the maintenance modes. All students completed these two tasks correctly.

8.5.4. Concurrency in embedded systems

While the back yard alarm task uses polling no investigation of timing effects relating to polling were noted. However while observing students during this task an issue with push-button switch action during single step mode was realised and led to significant changes with the switch model.

Two lecturers made reference to interrupt processing and how the visualiser could support understandings of how interrupt processing and polling work. One lecturer suggested extending the visualiser to expose I/O handshaking so that synchronisation issues could be explored.

8.5.5. The critical nature of embedded systems

Two of the exercises contained contexts that include aspects of criticality; the controlled pedestrian crossing is about safety of pedestrians and traffic flow, and the VOR Morse code task is about the safe landing of aircraft. No specific questions were asked of the students in these exercises that addressed criticality, however one student indicated a concern for criticality while undertaking the controlled pedestrian crossing task when he questioned the length of time given to pedestrians to cross the road and the time for vehicles to stop.

8.5.6. Programming understanding

The model chosen for this analysis was the five levels of programming thinking (Thuné & Eckerdal, 2009). In Appendix G the data collected while students undertook the tasks using the visualiser as well as transcribed comments have been identified and coded in relation to the programming thinking categories. The codes for surface thinking are ST-textual and ST-action; the codes for deep thinking are DT-application, DT-problem and DT-context.

8.5.7. Surface thinking (textual level)

After a simple introductory ‘hello world’ flashing LED program to introduce the visualiser all students undertook the Knightrider task. This required the addition of an extra three LEDs into a bi-directional sequence (Figure 8.7).

```
while (1) // repeat forever
{
    PORTA = 0b00000001;  //A.0
    _delay_ms (500);
    PORTA = 0b00000010;  //A.1
    _delay_ms (500);
    PORTA = 0b00000100;  //A.2
    _delay_ms (500);
    PORTA = 0b00001000;  //A.3
    _delay_ms (500);
    PORTA = 0b00010000;  //A.4
    _delay_ms (500);
    PORTA = 0b00001000;  //A.3
    _delay_ms (500);
    PORTA = 0b00000100;  //A.2
    _delay_ms (500);
    PORTA = 0b00000010;  //A.1
    _delay_ms (500);
}
```

Figure 8.7: Knightrider task starter program code

The most efficient solution to the exercise would be to insert the new code statements into the middle of the program sequence. This would indicate that students perceived the program as an application for an LED sequence rather than as lines of program code (textual thinking). Three of the students chose to add all the new commands to the end of the program line by line and then change all the lines of program code to build the correct sequence; they were seen as exhibiting a program textual level of programming thinking.

Student B was different and added the new lines into the middle of the sequence and then changed all the comments (not the commands) to reflect the correct sequence. This student then ran the program and after a brief period of time watching the visualisation realised that the commands needed to be changed as well as the comments. This indicated that student B was primarily operating at an application level of program understanding.

Textual thinking was also evident when students began a task by editing the program code without reading the instructions and then having to close the visualiser to read the instructions. Student A did this for the first four tasks and on the fifth read the instructions first, possibly indicating a progression from textual to application thinking. Student B always read the instructions first indicating his application thinking. Student C continually rejected encouragement to interpreting the instructions himself. While student D continually asked questions in the first tasks about what to do next, by the final task he interpreted the instructions himself without asking what to do.

8.5.8. Surface thinking at the action level

Action level program thinking is clearly evidenced by students when they see an error in the program operation and correctly change the code without trial and error or false starts. This level of thinking, while still at a surface level, represents a highly significant step in students' understanding and comprehension; it is the gap between structure and function. This is no easy gap for students, "in order to overcome this gap, learners have to take a run-up and jump – but they might not get the necessary momentum, as they fiddle with finding the correct steps, or they are even aiming at jumping in a wrong direction" (Schulte, 2008, p. 156).

In the results the data of interest is not the amount of action thinking by students but the difficulty experienced by students with action thinking. The data from this small sample reveals no false starts or random and inaccurate experiments on the part of students when modifying lines of code to achieve a purpose in the program. Some of this will relate to the fact that the students are not completely new to programming, however all students successfully modified lines of program code they had never experienced before in a context they had never experienced before and all commented on how easy it was to see what was happening.

Action thinking directly relates to the embedded system's property of hardware-software interaction and how the visualiser reveals the action of programming code so explicitly in the context of easy to understand and uncomplicated input / output devices. This ease with which students comprehend the function of lines of code may signal the most significant effect of the visualiser, as one lecturer commented, an "amazing use of context to help learning".

8.5.9. Deep thinking at the application level

This view of program code is a 'big-picture' or a top-down perspective of a program's operation. This requires 'know-how' knowledge and refers to being able to select and apply programming knowledge to solve a problem. This crucial aspect of teaching students how to translate problems into program code resounds throughout computer science education literature. Programming textbooks and introductory courses however continue to focus on declarative knowledge (syntax and semantics) rather than the more important aspect of learning schemas or patterns of programs and the strategies for how to put them together (Robins et al., 2003).

In the data this know-how was evidenced when students read, interpreted and translated instructions into program code, where they copied blocks of code and modified them or, after having run their code in the visualiser, changed a semantic error relating to program control

(syntax was correct but program operation was incorrect) and when they recognised for themselves that their program ran as per the instructions.

While application thinking could be exhibited in all the tasks, the VOR Morse code and the backyard alarm tasks required students to use application thinking. To support relational thinking between blocks of code, the flowchart examples (Figure 6.6, Figure 6.8 and Figure 6.9) were developed to support the strategies of merging and nesting code blocks (de Raadt, 2008a). These appeared to positively support students undertaking the task.

Students A and B independently used application thinking to complete the tasks, while students C and D required more assistance. Student D in particular struggled significantly with application thinking. He enclosed while(1) loops inside if statements (Figure 8.8) creating code that could never escape the loop. He asked what was wrong and instruction was given to him about the effect of a while(1) and to restructure the code back into a single while(1) main loop; then he was walked through the flowchart. After this the student still continued to struggle with application thinking unable to see a series of redundant if statements in the code controlled by the main on/off switch (Figure 8.9). After further questioning the student realised their effect and removed them.

```
if AlarmOnOfSw_IsLow();
{
    while(1)
    {
        if (PoolShedSw_IsLow())
        {
            set_PoolShedLed();
        }
        if (GarageSideDoorSw_IsLow())
        {
            set_GarageDoorLed();
        }
        if (ShedDoorSw_IsLow())
        {
            set_ShedLED();
        }
    }
}
if AlarmOnOfSw_IsHigh();
{
    while(1)
    {
        if (PoolShedSw_IsHigh())
        {
            set_PoolShedLed();
        }
        if (GarageSideDoorSw_IsHigh())
        {
            set_GarageDoorLed();
        }
        if (ShedDoorSw_IsLow())
        {
            set_ShedLED();
        }
    }
}
```

Figure 8.8: Student D non-escaping while(1) – non-application thinking

```
else if (AlarmOnOfSw_IsHigh())

  if (PoolShedSw_IsLow())
    {
        reset_PoolShedLed();
    }
    if (GarageSideDoorSw_IsLow())
    {
        reset_GarageDoorLed();
    }
    if (ShedDoorSw_IsLow())
    {
        reset_ShedLED();
    }
```

Figure 8.9: Student D unnecessary if statements – non-application thinking

From this testing it was evident that even with the support of the flowcharts two students were confused by the complexity of its logic in the backyard alarm task.

8.5.10. Deep thinking at the problem level

While problem thinking was not a requirement of the tasks, one student exhibited problem thinking in the controlled pedestrian crossing task by asking about timing to allow pedestrians to cross the road.

8.5.11. Deep thinking at the context level

No students showed evidence of programming thinking at the deepest level of context. This is where programming understanding extends beyond the program into the student's world and has potential for helping them later in life (Eckerdal & Berglund, 2005).

8.6. Changes indicated from user interaction with the visualiser

While a number of changes to tasks, the visualiser and System Designer were indicated from testing, only the ones significant to student learning are reported here.

8.6.1. Results indicating changes needed to tasks

During all tasks, but specifically the backyard alarm and VOR Morse code tasks the number of times students had to change what they were viewing on the screen to read and re-read instructions was considered as excessive and did not follow good layout practices in terms of cognitive processing by students.

8.6.2. Results indicating changes needed to the visualiser

During testing of the VOR Morse code task it was noticed that a significant amount of jumping around in the code window occurred when it came to control moving from the main loop to the code in functions. This was seen as affecting the coherency aspect of the visualiser model.

Whilst developing the visualiser a standalone visualiser option was added to System Designer under the Visualisations menu. It was envisaged that this would be removed once testing was completed. One lecturer indicated that his should remain as it provides a simpler interaction with the visualiser when undertaking learning tasks that have no input or output but only memory use.

While testing with student A it was realised that the models used for push buttons were not going to be satisfactory while single stepping through code, and that features for different switch types would also require different visualisation characteristics during the single step and run modes of the visualiser so as to support understandings of concurrency. During testing for the other three participants the push button switch was changed to a toggle switch so as not to confuse the students. This was seen as significant as it relates to embedded systems state.

To support students with syntax System Designer incorporates auto-generation of program code and the live coding of flowchart elements (Figure 8.10). As flowchart elements are added to or removed from the left hand side of the diagram the code window on the right hand side is

automatically updated. While this was used by all students, Student B struggled with syntax in the backyard alarm task with structuring the braces for the code for an *if(){}-else{}* even after using the flowchart model.

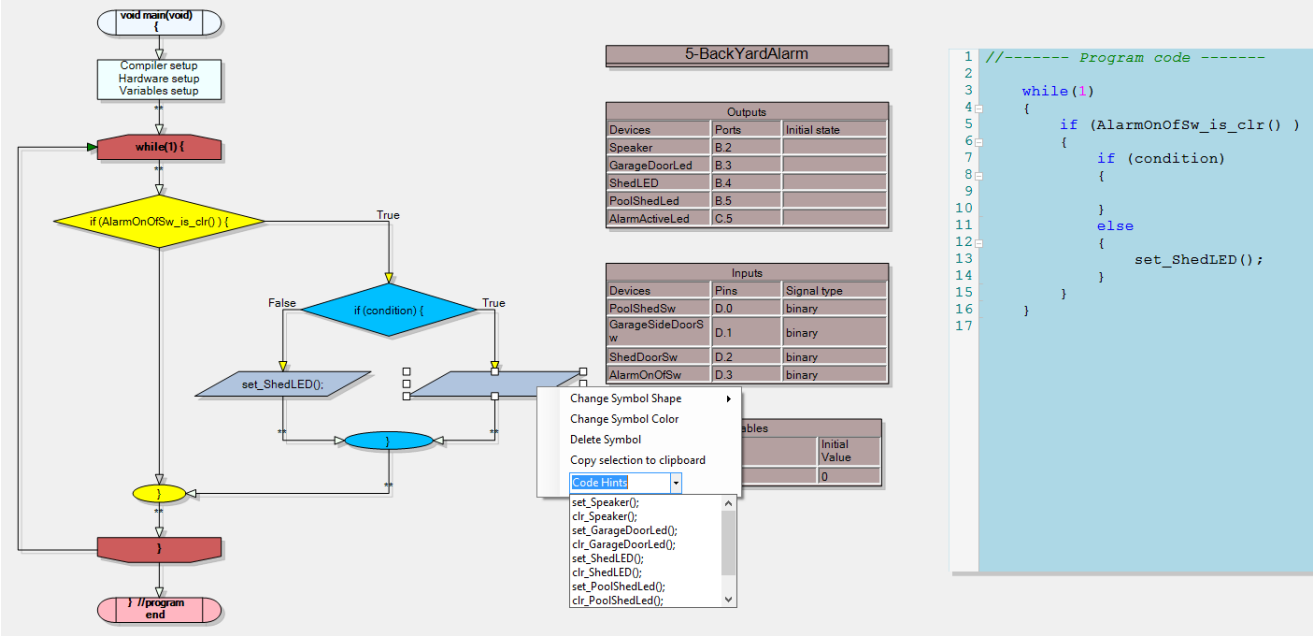


Figure 8.10: Live coding feature of the flowchart in the backyard alarm task

8.6.3. Changes to System Designer

Testing of the application led to the observation that when adding new components to the diagram a significant amount of work was required by all four students to get the component into the desired position on the diagram and impacted upon the application's ease of use.

Chapter 9. Discussion

In this chapter the degree to which the results support each research question is discussed. Research question one, which investigated the success of the visualiser as a model and an analogy; is discussed in section 9.1. The second research question, which investigated the visualiser's benefits for learners of embedded systems, is discussed in section 9.2 in terms of the feedback that it provides learners, its support for developing a notional machine, and its ability to scaffold syntax learnings for students. The limitations of the study are presented in section 9.3 and future directions are covered in section 9.4.

9.1. RQ1: the visualisers success as a model based learning environment

9.1.1. The visualiser as a model

The visualiser received very positive comments from both students and lecturers with regard to the seven characteristics of good models as presented in section 8.4. It became clear however, that as a model it cannot be isolated from the tasks students undertake with it. Specifically it was observed during several exercises that all students had to move backwards and forwards a number of times between the instructions and code window as they read and re-read instructions. This is counterproductive for learning as it increases representational holding. Representational holding is where the learner exceeds the cognitive capacity of working memory by having to maintain the essential processing of what the visualisation is doing while also holding information about the task (Mayer & Moreno, 2003, 2010). Where students had to move backwards and forwards between the visualiser and the task instructions it worked against the completeness and consideration aspects of the whole learning process. The layouts for these exercises have been reworked to reduce representational holding demands.

The second situation in which cognitive load became an issue was in the VOR Morse code task. In this task the repeated calling of functions became confusing for students as the code window moved a great deal while code was being run. This reduced the conciseness and conceptual aspects of the visualiser because of the increased cognitive load.

To combat this a new feature has been developed which gives the user the ability to skip visualisation (but not execution) of functions. In the VOR Morse code task the functions dot and dash have the commands *–-noviz* and *–-notrace* as comments in the function declaration (see

lines 58 and 67 in Figure 9.1). In this way a user can test a function and then, once satisfied with its working, can add a command that instructs the visualiser to process the code but not visualise the line by line action of the function. This reduces the movement in the code window and increases the conciseness of the visualiser by reducing extraneous processing. This relates the visualiser back to a primary purpose of modelling which is to be able to test and trial ideas quickly and efficiently (France, Compton, & Gilbert, 2010; Mayer, 1989). If students are confused by the visualiser they will not use it to its fullest potential.

```
38  while(1)
39  {
40      sendWP();
41      _delay_ms(5000)
42  }
43
44  //function to send the two letters WP for Whenuapai
45  void sendWP() //  W= .--  P = .--.
46  {
47      dot();
48      dash();
49      dash();
50      _delay_ms(800);
51      dot();
52      dash();
53      dash();
54      dot();
55  }
56
57  //function to send a dot (short beep)
58  void dot()  //  --noviz --notrace
59  {
60      set_MORSE_LED();
61      sound (PORTB, B1, 300, 500);
62      reset_MORSE_LED();
63      _delay_ms(200);
64  }
65
66  //function to send a dash (long beep
67  void dash() //  --noviz --notrace
68  {
69      set_MORSE_LED();
70      sound (PORTB, B1, 1000, 500);
71      reset_MORSE_LED();
72      _delay_ms(200);
73  }
```

Figure 9.1: Visualiser -noviz and -notrace commands

9.1.2. The visualiser as an analogy

The understanding that a model is an analogy or metaphor and therefore has limitations in its representation of reality is clearly recognised in the literature; the significance for teachers is that novices and experts differ in their appreciation of this metaphor. While experts understand that a model is only suitable for testing specific ideas, novices ascribe a general reality to a model that it does not warrant (Coll et al., 2005).

This was observed clearly with student C in the EMDR light bar task, where he was not able to separate the delays observed in the visualiser between those caused by the step delay of the visualiser and those caused by the delay commands in program code (section 8.4.7.1). In being unable to separate these he thought that the visualisation should be equally real when single stepping the code, running the code at slow speed and also running the code at full speed. He consequently became confused when the visualisation appeared correct at slow speed but not at full speed and when questioned about this he could not identify why one appeared correct and the other did not, stating that the program was correct even though it was not. Student A and student D had no issue with the delays, however student B provides insight into this issue as he was initially confused but bypassed his confusion by combining two lines of code onto one line in the visualiser (as in Figure 8.4).

This analysis of the visualiser's limitation has significant implications with regard to learning about reactivity and concurrent processing in embedded systems, and having strategies in place to counter these effects. As described by Kaila et al. (2011) pre-training with a visualisation is important to reduce the cognitive load of students while they learn a tool. However this result has clarified a further aspect of training with regard to the limitations of the analogy. Students must know the differences between running code at full speed, slow speed and single step modes and that only running code at full speed will allow complete investigation of real-time effects. As part of this the strategy used by student B, to combine lines of code onto one line in the program, can be used to mitigate the effect in some situations. Changes to the visualiser have also been made: two new options will be trialled further with students for single stepping code and slow running of code. The first is the option that delay commands will be stepped over (not visualised at all) and the second is that delay commands will be shown but the delay will not be processed.

For teachers the significance is that many students will be unable to separate the analogy from real hardware without assistance. They must do this if the model is to fulfil its purpose of helping them grow metacognitively through trialling, reflecting on and discussing their ideas about real hardware (Coll et al., 2005), which means accurately learning the reality that the visualiser represents and not the model.

9.2. RQ2: the visualisers benefits for novice learners

While the first question is specific about the visualiser as a model-based learning tool, the second question about the efficacy of the visualiser is general and relies on understandings from the

literature review to identify the meaning of the results. These include: the importance of feedback, the development of a viable notional machine, known novice programming issues, visualisation, learning theory, the role of the task, structure versus function, embedded systems concepts and variation theory.

9.2.1. The visualiser as a feedback device for student learning

One of the important aspects of learning in embedded systems when using hardware is the meagre amount of feedback novices gain about what is happening inside an embedded system (see Figure 3.1) as described by Cho (2009). Feedback however is crucial as it is one of the most powerful and important tools for learning (Hattie & Timperley, 2007; Hattie, 2011). While the visualiser models many of the hidden processes, thus presenting students with more feedback than a hardware only environment, this needs discussing in detail so that users are aware of the strengths and limitations of the visualiser as a feedback tool. To a learner feedback must relate to their performance (Hattie & Timperley, 2007) and needs to be: responsive, relevant, specific, positive and frequent, but not too frequent (Alton-Lee, 2003).

The visualiser provides participants with responsive feedback at both surface and deep levels of program understanding. At the surface level the lines of code may be either correct or incorrect as indicated by output devices or recognition of input devices. At deeper levels the operation of the system is also feedback; however it can be negated by students' understanding of the task. This was observed when participants did not make sufficient effort to interpret and understand the task before attempting it and thus did not know whether they had it correct or not. A teacher's input is crucial in making sure students are clear about task objectives.

Relevance is a strong feedback feature of the visualiser as evidenced by the ease, directness and accuracy with which all students carried out almost all tasks. The feedback however is not always directly relevant to the error and requires some indirect understanding. This was the case with the incorrect data direction register configuration in the EMDR light bar task (an error which has been observed often in class). Students need the teacher to train them in interpreting feedback as sometimes the error will not relate to what appears to be the obvious line of code. The teachers intervention with the simple question, "what else has an influence on this output/input/etc?" is crucial in keeping the student on the right path.

Specific and precise feedback to users is a feature of the de-cluttered nature of the visualiser's interface. It is central to the controlling of the amount of information presented to users by

starting with only the code and error windows and being able to show or hide other windows as required. While this was not directly tested by using different views with different users it was confirmed by the direct way in which students identified the information they needed to carry out tasks.

Feedback can be positive or negative when it is linked to how students view their own abilities as learners, either as competent or not. While feedback from the visualiser is inherently neutral, a student's capability in interpreting the feedback will control whether it is has a positive or negative effect. In general the participants appeared to have interpreted their time using the visualiser as a positive experience, as throughout most tasks they had little difficulty interpreting the feedback from the visualiser and none showed evidence of duress or frustration during testing.

Learning can be subverted by too little or too much feedback, and a balance is required. With too little feedback the learner will not be able to differentiate what it is they need to learn; this directly relates to the goal of the visualiser in demystifying the hidden operation of an embedded system. As students managed the debugging of unfamiliar embedded systems program code with little difficulty it appears that the amount of feedback from the interfaces was sufficient but not excessive.

While embedded system hardware reveals too little feedback, compilers often provide too much feedback, as a simple error in code can lead to a significant number of compiler messages. A novice learner's ability to interpret obscure compiler error messages is very limited (Ahmadzadeh et al., 2005; McCauley et al., 2008; Murphy et al., 2008; Simon et al., 2008; Vessey, 1985), and provides them with an overwhelming amount of feedback. In this study none of the students attended to the error window when there were errors without being prompted at least once. This result does not reveal why students ignored the errors, it could be that they were afraid of the errors, that they did not notice the error window or there was another reason, so no conclusion about error feedback is made. While students did not attend to the error window it was still observed that the parser generated several lines of error messages on some single errors. Subsequently the parser has been modified to stop on some errors. While this is not recommended practice in compiler writing (Fischer, Cytron, & LeBlanc, 2009; Mak, 1996; Torczon & Cooper, 2007) it was deemed important for supporting novice learners to reduce the extraneous processing. The result that no student attended to the error window is also of concern and the visualiser has been changed to now include an audible as well as visual indication that an

error exists. This will direct the user to attend to the error and can be used in future testing to help understand how the user responds to and interprets the visualiser's error messages.

Testing of the visualiser provided a significant understanding of the limitations of the visualiser as a feedback tool for learners; and while the tool has much power and it is evident that it can positively support the learning process it could never completely replace the teacher as a provider of feedback.

9.2.2. The visualiser as supporting development of a notional machine

When learning using a compiler and microcontroller hardware, once the code is successfully compiled it disappears into the microcontroller. When this happens novice learners are unclear on the effects of lines of code so they are unable to form viable models of its operation. When the user can maintain these understandings they have created their own notional machine - a view of how the programming language works and what is going on inside the computer (du Boulay, 1986). While visualisation is well known to support this in terms of learning to program, several results carry this over into evidence of development of a viable mental model of the characteristics of embedded systems as well.

The results show quite clearly how students were able to quickly grasp and then consistently show thinking about hardware-software interactions for previously unfamiliar hardware input and output commands. They did this by quickly recognising and correcting semantic errors, and then carrying those understandings with them into other programs without error. This is an example of variation theory where repetitive slight variations are encountered in sufficient quantity to allow the learner to generalise an understanding (Marton, Tsui, & Runesson, 2004; Thuné & Eckerdal, 2009). The fact that three of the students were able to generalise this learning by resolving the data direction register error showed their development of a viable mental model of hardware-software interaction. While it was clear that student C had developed a notional machine it was at a lower level as it did not allow him to see the indirect hardware-software interaction.

With regard to the reactive nature of embedded systems the results show clearly how two students gained a viable understanding of this. Students A and D had similar viable mental models of much program code as they showed no identifiable programming misconceptions with the level of C code used in the exercises. Both however made one similar mistake: Student A removed the while(1) in the VOR Morse code task, and student D created two instances of

while(1) inside separate if() statements in the backyard alarm task. Both quickly realised their errors; student A corrected it himself, student D asked what the while(1) meant and then went on to correct his error independently.

The indications here in terms of notional machines are that initially student D did not have as strong a model of the while() statement as student A. However neither had a sufficient concept of the while(1) prior to the exercise to stop them removing it. The fact they had seen this code pattern in the prior tasks did not present enough generalisation in terms of variation theory for them to develop a clear model of its meaning. It was because they made the error that they experienced a contrast pattern in terms of the while(1) and the fact that they corrected their errors themselves showed their new strong model of it. Significantly this result is not about a notional machine for a while() statement but about a notional machine for reactivity of embedded systems; as the key reason for the while(1) is, as student A said, to "continually run it[the program] through". These two students now have a clear understanding of embedded systems as reactive rather than transactional in nature. One implication from this is that while a generalisation pattern of variation was sufficiently present to help students develop some concepts, a contrast pattern of variation is useful to learn concepts that are not as obvious.

Literature on learning has some general findings such as students needing sufficient opportunities to learn (Alton-Lee, 2003), that simply seeing a visualisation provides no benefit for learners (Coll et al., 2005), and that active engagement with the visualiser is crucial (Hundhausen et al., 2002). Variation theory enhances these general findings by providing clear strategies to support them and these results support this by revealing how much variation is enough for student learning.

A notional machine is also part of the important understanding of state in embedded systems. Students must understand that a system is not transactional but that its condition is influenced by what has happened before and that its present condition impacts its future condition. This is identified at two levels in this study. The first is the code or action level as shown in the EMDR light bar and the pedestrian crossing tasks where LEDs need to be not just turned on but off as well. Student C initially struggled with this actually saying that the delay command should turn off the LED; this bizarre type of thinking is not uncommon in students (Pea, 1986). He built a more comprehensive notional machine about state through this exercise as he had no further issues with it in later exercises and used terminology more correctly.

The second level of state is the machine or application level as in the pedestrian crossing task with its normal and pedestrian crossing states, the backyard alarm task with its armed and unarmed states and the VOR Morse code task with its normal and maintenance states. While the students experienced sufficient variation to gain an understanding of the action level of state, it is not possible to suggest that they have had enough exposure to machine state as all their tasks only had two states. Two states will not provide the variation needed to explore the state concept fully.

In the same way the real-time aspects of embedded systems were present in tasks via the backyard alarm task where three switches are polled to see if one of them has been activated. Extrapolating results about variation from a single application is not possible though.

The results identified the limitations of visualiser use on several aspects of embedded systems learning. This could equally be expressed as the distinction of effect between task and visualiser. It was noted that hardware-software interaction, reactivity and the action level of state are inherent when engaging with the visualiser and there is less effect by the task. The real-time, application level of state and critical aspects of embedded systems however were not noted from the tasks students undertook and therefore need carefully contextualised tasks. For instance the pedestrian crossing exercise needs students to engage with safety of pedestrians to help them learn the critical aspects of embedded systems and extending to a full intersection to explore state. Also the backyard alarm and VOR Morse code tasks need extra states with more complex interactions added to them.

While real life contexts have always been used in class exercises the amount and specific type of variation required by students to learn new concepts needs further consideration. Because using actual hardware takes lengthy amounts of time, having more exercises has not been possible in class. The addition of the visualiser should resolve this as it is a quick tool to use with minimal setup.

9.2.3. Syntax issues

During testing, student B struggled with the syntax for embedded if() statements; and had particular trouble with the layout of braces. This is not a surprising result as it is not uncommon to find students struggling with syntax at a level unanticipated by teachers (Denny et al., 2011). While System Designer already incorporates a number of syntax supports for students through automatic generation of program code for embedded systems hardware and automatic generation

of program code for flowcharts and state machines, a further set of syntax supports has now been added in the form of popup code snippets.

The code snippets for a switch() are shown in Figure 9.2 and those for a for() are shown in Figure 9.3. As code is entered into the editor the first drop down will appear giving a choice of examples of the code to use; selecting one of these will open the second drop down which shows the actual layout as well. Clicking on the code snippet will insert it into the program. These code snippets have been developed to not just provide templates but more complete examples of program code to support learners as in shown in Figure 9.3 with the for().

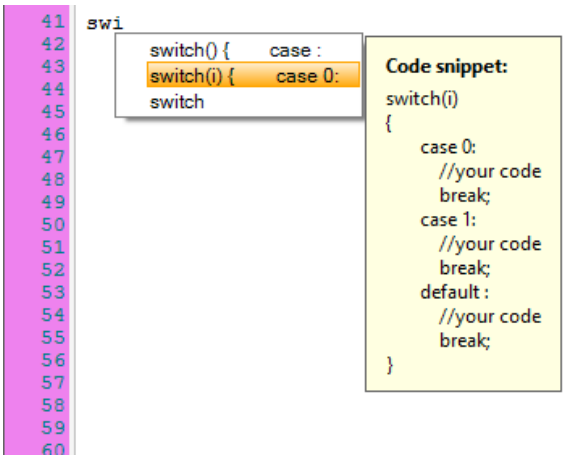


Figure 9.2: Code snippet for 'switch'

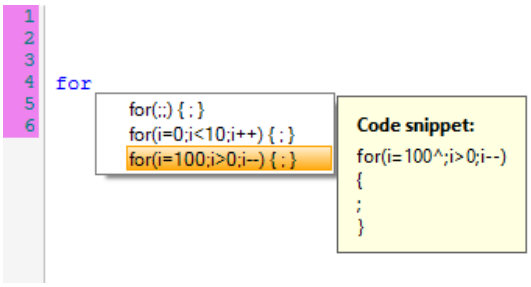


Figure 9.3: Code snippet for 'for'

9.2.4. Processes assessed in technology education

As discussed in section 2.2, technology education relates to process based understandings rather than being content driven and students are assessed on the process (called technological practice) that they undertake when developing a product. The goal of this process is to create a project which is fit for purpose in the broadest possible sense, which includes both the technical requirements and the social (stakeholder) requirements. The speed and ease with which participants were able to alter and then dynamically view their work indicates that students in the classroom should be able to do this with their project's stakeholders (with appropriate training by the teacher). The ability to model a system before committing it to hardware means that a closer relationship with stakeholders' needs should become evident as students use stakeholder feedback to quickly inform their designs. This will result in them being able to more completely meet the stakeholder relationship requirements of their assessments.

9.3. Limitations of the study

The study used a small number of participants who were selected from a specific cohort of learners; so generalisability of results was not an outcome of the study. Also the data collection and analysis methods, while allowing close inspection of learning details also do not provide the opportunity to generalise results. Such qualitative research methods place the onus on the reader to determine applicability to their own context (Case & Light, 2011).

Tight selection of participants to students with some programming experience means that the results will differ when the tool is used with programming novices. The significant issue with this is that when introducing the visualiser to students this should be done in conjunction with real hardware so that students learn about the real hardware that the model represents and not just about the model (Coll et al., 2005).

9.4. Future research

A number of research objectives are being developed and others will be investigated to explore the efficacy of the visualiser as a learning environment for novice learners in secondary schools.

While System Designer is used with students, the visualiser extension has not been implemented into the actual learning program as yet. Therefore the understandings that students develop while using the visualiser have not been tested as to whether they enhance students' understanding about real embedded systems hardware. This is the first analysis that will be undertaken with students. The tool will be introduced to two non-specialist electronics teachers for their use with students and results compared against other students who will not use the visualiser.

As students use the visualiser to solve issues relating to their projects data will be collected as to how the visualiser supports the processes of technological practice and technological modelling.

The need for embedded systems to react and respond within the time constraints of their environment has not been explored in this study and requires careful exploration. This requires the development of tasks that engage students with sufficient variation through problems of timing, using for example the maintenance of traffic flows through busy intersections, management of dynamic manufacturing processes or control of fast acting devices such as pacemakers. Real-time understandings also require interfaces for the visualiser that allow accurate simulation of real world timing. A pulse input interface (Figure 9.4) has been developed that allows tasks to have repetitive timed inputs. When completed it will have the facility of

dynamically changing its timings to reflect whether the visualiser is in run, slow or single step mode. This will require testing with students to determine whether it mitigates the issue student B experienced, has no effect or even confuses students further.

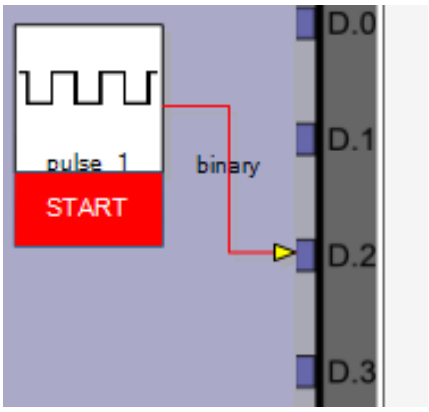


Figure 9.4: Pulse input interface for real-time learning

The results that relate to variation theory are not specific enough and require quantifying in terms of how much variation is enough to learn a specific concept. This will be specifically tested with students.

Currently students take static tests and examinations at school, where they fix or complete snippets of program code. This environment relies on rote learning and while it reveals some understandings of students it does not allow students the opportunity to present their full understandings such as interpreting and debugging dynamic code issues or understanding how program plans can be combined. The use of the visualiser in summative testing will be trialled in the future and will cover aspects such as error correction of dynamic problems and simple programming tasks that require the combination of strategies.

While the complex nature of C as a first programming language means that it will not replace BASIC currently for junior students, the development of the visualiser gives the opportunity to introduce C to senior students (year 12 and year 13). The C language will be taught to these students using only the visualiser, leveraging off their familiar understandings of embedded systems. Students will then be given the choice of using BASIC or C for their project work.

Whilst the visualiser is freely available as a tool it is not currently used by other schools. A colleague at another school has requested training on the visualiser and if he uses it with his students he is willing to collect data on its use.

9.4.1. Future changes to the visualiser and the learning program

Current development of the visualiser involves a hardware timer class and the integration of an oscilloscope library to view I/O ports and individual register bits. These are at a preliminary stage. Once completed these will be trialled with students in their 3rd and 4th years at secondary school to model timing and PWM.

Error handling within the visualiser will be modified in the future. The aim is to provide more immediate feedback to students as they enter their code; possibly with the integration of automatic correction of some syntax errors.

Some students have previously taught themselves C++, and I am very interested in object oriented understandings. System Designer contains a diagram tool for automatic class generation with the ability to automatically generate C++ code templates for classes; as well as this it has a sequence diagram tool. The further development of this would be to integrate it with the Context Diagram and State Machine Diagram tools. This would allow more than skeleton code for the class to be generated, System Designer would then embody a fuller model-based software design environment for simple embedded systems using an MVC model. The visualiser will be extended to parse C++ classes as well.

Chapter 10. Summary

This research is part of the ongoing development of resources for school students and colleagues in learning about embedded systems and was triggered by continuing concerns with student misconceptions about programming understandings of embedded systems. With regard to the programming misconceptions of students Roy Pea said that we should not be amazed at the "bizarre behaviour" (Pea, 1986, p. 33) of our students, but anticipate it. This study aimed to achieve just that.

A wide ranging and detailed investigation into relevant literature was undertaken to inform the issues with learning in the classroom. It drew from both secondary and tertiary educational literature beginning with an investigation into learning theory, best pedagogical practices, engineering education, the New Zealand Curriculum and the paradigm change in technology education from its behaviourist roots to the new participatory metaphor. These understandings were extended with literature on model-based learning environments, cognitive load and the use of conceptual models in education.

Action research findings from single purpose visualisers I had previously developed for use with students were presented to justify the research effort. This was followed by an analysis of selected embedded systems simulation and programming learning environments in terms of best practices for model-based learning to establish the best possible feature set for a novice friendly visualiser for an embedded system.

Subsequently lexers and parsers for interpreters for both C and BASIC programming languages were written. BASIC was chosen because it has benefits for novices while C is well established in industry and engineering education; so both were deemed important to the visualiser's success. The existing application, System Designer, that students use to plan and record their project work on ATMEL AVR microcontroller based embedded systems was augmented with a visualiser using these interpreters. Not all language features of C and BASIC are fully implemented in the visualiser, and while its features may grow in the future the goal is not to produce a production tool but a pedagogical tool for students and colleagues. The visualiser is designed to ultimately become redundant once students have developed the concepts they need; students who continue in this field of learning will need to transition into using professional tools.

Two research questions were investigated. The first involved the visualiser as a model based learning environment. The second related to the effects of the visualiser on novice learners. To answer these questions a small group of students were selected and tested in a qualitative study. They were asked to carry out a range of tasks, which were recorded using screen capture software and also to give final comments on the visualiser's characteristics. These students all had the same programming background: two semesters of learning and no other programming experience. None had experience with embedded systems. Tertiary lecturers had the visualiser demonstrated to them and were asked to comment on it as well, using a semi-structured interview process.

The student tasks was analysed and coded to identify data that related to the visualiser's model-based learning characteristics, learning about embedded systems and students programming understandings. Along with this any changes to tasks, the visualiser and System Designer were noted, as well as errors with the C parser. All of which were used to inform the visualiser's final development before implementation into the school learning program.

The participants' interactions with the visualiser provided evidence for six of the seven characteristics of good models showing it to be: complete, concise, coherent, concrete, conceptual and considerate. The last characteristic of correctness was inferred from lecturers' comments and recommendations. The visualiser is an analogy of an embedded system; as such it acts like an embedded system in many but not all respects. When the visualiser is in single step or slow running mode then time related code may appear to be correct when it is not, as experienced by one participant. This limitation while obvious to an expert is not at all obvious to a novice learner, who may confuse what is happening in the visualiser with what happens in a real system. While modifications to the visualiser were made to counter some of this effect, teachers must work within this limitation, explaining it to students and not expect students to comprehend it for themselves on first use.

For learning about the hidden or black box nature of embedded systems the visualiser showed significant benefits in terms of the mystical concept of interrelatedness between hardware and software, the reactive nature of embedded systems and the concept of state. Some of these understandings are inherent when students engage with the visualiser; others require students' engagement with strategic exercises that elicit these understandings. Contextualised tasks are one of the ways this can happen; the other is via sequences of exercises that bring about

understandings through one of the four methods of variation theory. Exploration of the real-time and concurrent aspects of programming embedded systems is still required.

Because students at school have not taken any programming classes before they take classes in embedded systems, the assessment of programming understandings was an essential aspect of this research as well. A model of programming thinking was used in the data analysis to identify the surface and deep understandings that the participants exhibited while using the visualiser. The results of testing showed how the visualiser and the tasks used with students each contribute to understanding. While the visualiser showed clear results for the surface action and deeper application levels of program thinking, development of still deeper understandings requires students to undertake tasks with real life contexts.

The visualiser was found to allow students the ability to quickly and easily trial their ideas, something school students are not able to do with real hardware in class as they may need to reconstruct parts of their hardware before each trial. In class the visualiser is anticipated to have significant advantages for students as: it reduces the amount of time required to come to modify a solution, encouraging students to actually modify their work to achieve the best solution rather than accept the one they have already constructed. Significantly it will allow students the opportunity to interact with end users much earlier in the process of developing an outcome, which more closely meets the participatory learning paradigm underpinning technology education. Because the visualiser is quick to use, the teacher has the opportunity to provide more learning situations for students; if strategically selected tasks are used this has significance in terms of variation theory as it creates sufficient variance for student understanding to develop.

This study has been professionally rewarding; deepening my pedagogical content knowledge, that is the interrelationship between understandings of educational practice and the content of embedded systems and learning to program. Specifically this relates to the collection of data; this was a highly informing process compared to that usually undertaken at school, where a task is developed and trialled with students, and afterwards is modified but remains untested until the next cycle of students takes the course a year later. The opportunity to observe student understandings and then analyse them in such a comprehensive and detailed way revealed clearer understandings about student learning than classroom practice has.

References

Abrantes, J., Porter, A., Meyers, W., Stace, R., Susilo, W., Krishna, A., … Xia, T. (2008). CSCI: A LEAP into the future. In *Emerging Technologies Conference 2008* (p. 1).

Adamo, O. B., Guturu, P., & Varanasi, M. R. (2009). An innovative method of teaching digital system design in an undergraduate electrical and computer engineering curriculum. In *Microelectronic Systems Education, 2009. MSE'09. IEEE International Conference on* (pp. 25–28). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5270837

Adcock, B., Bucci, P., Heym, W. D., Hollingsworth, J. E., Long, T., & Weide, B. W. (2007). Which pointer errors do students make? In *ACM SIGCSE Bulletin* (Vol. 39, pp. 9–13). Retrieved from http://dl.acm.org/citation.cfm?id=1227317

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In *ACM SIGCSE Bulletin* (Vol. 37, pp. 84–88). Retrieved from http://dl.acm.org/citation.cfm?id=1067472

Al-Barakati, N. M., & Al-Aama, A. Y. (2009). The effect of visualizing roles of variables on student performance in an introductory programming course. In *ACM SIGCSE Bulletin* (Vol. 41, pp. 228–232).

Alton-Lee, A. (2003). *Quality teaching for diverse students in schooling: Best evidence synthesis June 2003*. Wellington, New Zealand: Ministry of Education. Retrieved from http://intra.bay.net.nz/Learning_objects/datas/quality.pdf

Anderson, L., Blumenfeld, P., Pintrich, P. R., Clark, C. M., Marx, R. W., & Peterson, P. (1995). Educational psychology for teachers: Reforming our courses, rethinking our roles. *Educational Psychologist*, *30*(3), 143–157. doi:10.1207/s15326985ep3003_5

Arends, D., & Kilcher, A. (2010). *Teaching for student learning: Becoming an accomplished teacher*. Routledge.

References

Atmel Corporation. (2013a). Atmel Studio. *Atmel*. Retrieved March 12, 2013, from http://www.atmel.com/tools/atmelstudio.aspx?tab=overview

Atmel Corporation. (2013b). Using Simulator in Atmel Studio. Retrieved February 12, 2012, from http://www.atmel.no/webdoc/simulator/simulator.wb_Simulator_Use.html

Avalon Dock. (2013). Avalon Dock. *Avalon Dock*. Retrieved March 12, 2013, from http://avalondock.codeplex.com/

Baldwin, L. P., & Kuljis, J. (2004). Visualisation techniques for learning and teaching programming. *Journal of Computing and Information Technology*, *8*(4), 285–291.

Ball, T., & Eick, S. G. (1996). Software visualization in the large. *Computer*, *29*(4), 33–43.

Barak, M. (2009). Motivating self-regulated learning in technology education. *International Journal of Technology and Design Education*, *20*(4), 381–401. doi:10.1007/s10798-009-9092-x

Barnett, R., O'Cull, L., & Cox, S. (2007). *Embedded C Programming and the Atmel AVR*. Cengage Learning.

Ben-Ari, M. (1998). Constructivism in computer science education. In *ACM SIGCSE Bulletin* (Vol. 30, pp. 257–261). Retrieved from http://dl.acm.org/citation.cfm?id=273133.274308

Ben-Ari, M. (2004). Situated learning in computer science education. *Computer Science Education*, *14*(2), 85–100.

Ben-Ari, M. (2005). Situated Learning in "This High-Technology World." *Science & Education*, *14*(3-5), 367–376. doi:10.1007/s11191-004-7934-1

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N., & Sutinen, E. (2011). A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, *22*(5), 375–384.

Ben-Ari, M., Berglund, A., Booth, S., & Holmboe, C. (2004). What do we mean by theoretically sound research in computer science education? *ACM SIGCSE Bulletin*, *36*(3), 230–231.

Ben-Ari, M., & Sajaniemi, J. (2004). Roles of variables as seen by CS educators. *ACM SIGCSE Bulletin*, *36*(3), 52–56.

References

Bennedsen, J., & Caspersen, M. (2008). Exposing the programming process. *Reflections on the Teaching of Programming*, 6–16.

Benveniste, A., & Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, *79*(9), 1270–1282.

Berendsen, Y., Krammer, H., & Onderwijskunde, T. (1992). Problem decomposition using programming plans. *Tijdschrift Voor Didactiek Der B-Wetenschappen*, (3), 178.

Bertels, P., D'Haene, M., Degryse, T., & Stroobandt, D. (2009). Teaching skills and concepts for embedded systems design. *SIGBED Rev*, *6*(4), 1–4. doi:10.1145/1534480.1534484

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning*. Academic Press New York.

Bignami. (2013). Luigi. *MSDN*. Retrieved April 16, 2013, from http://web.tiscali.it/ggbhome/umlpad/umlpad.htm

Blumenfeld, P. C., Soloway, E., Marx, R. W., Krajcik, J. S., Guzdial, M., & Palincsar, A. (1991). Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, *26*(3-4), 369–398.

Bohn, R. E. (1994). Measuring and managing technological knowledge. *Sloan Management Review*, *36*, 61–61.

Booth, S. (1997). On Phenomenography, Learning and Teaching. *Higher Education Research & Development*, *16*(2), 135–158. doi:10.1080/0729436970160203

Booth, S. (2001). Learning computer science and engineering in context. *Computer Science Education*, *11*(3), 169–188.

Boustedt, J. (2008). A methodology for exploring students' experiences and interaction with large-scale software through role-play and phenomenography. In *Proceedings of the Fourth international Workshop on Computing Education Research* (pp. 27–38). Retrieved from http://dl.acm.org/citation.cfm?id=1404524

References

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? In *ACM SIGCSE Bulletin* (Vol. 39, pp. 504–508). Retrieved from http://dl.acm.org/citation.cfm?id=1227482

Bradbeer, J. (2006). Threshold concepts within the disciplines. *Planet Special Issue on Threshold Concepts and Troublesome Knowledge*, 16.

Broman, D. (2010). Should Software Engineering Projects Be the Backbone or the Tail of Computing Curricula? (pp. 153–156). IEEE. doi:10.1109/CSEET.2010.35

Broman, D., & Sandahl, K. (2011). How can we make software engineering text books well-founded, up-to-date, and accessible to students? In *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference on* (pp. 386–390). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5876110

Broman, D., Sandahl, K., & Abu Baker, M. (2012). The Company Approach to Software Engineering Project Courses. *IEEE Transactions on Education*, *55*(4), 445–452. doi:10.1109/TE.2012.2187208

Brown, J. S., Collins, A., & Duguid, P. (1989). Situated cognition and the culture of learning. *Educational Researcher*, *18*(1), 32–42.

Broy, M., & Stauner, T. (1999). Requirements engineering for embedded systems. *Informationstechnik Und Technische Informatik*, *41*, 7–11.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2006). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Transforming IT Education: Promoting a Culture of Excellence*, 301–325.

Brusilovsky, P., & Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. *Journal on Educational Resources in Computing (JERIC)*, *5*(3), 6.

Byckling, P., & Sajaniemi, J. (2006). Roles of variables and programming skills improvement. *ACM SIGCSE Bulletin*, *38*(1), 413–417.

References

C# Dictionary vs. List. (2013). *dotnetperls.com*. Retrieved April 11, 2013, from http://www.dotnetperls.com/dictionary-time

CAÑAsf, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human Computer Studies*, *40*(5), 795–811.

Carifio, J., & Perla, R. J. (2007). Ten common misunderstandings, misconceptions, persistent myths and urban legends about Likert scales and Likert response formats and their antidotes. *Journal of Social Sciences*, *3*(3), 106.

Carlisle, M. (2013). RAPTOR. *RAPTOR*. Retrieved March 12, 2013, from http://raptor.martincarlisle.com/

Carstensen, A. K., & Bernhard, J. (2007). Threshold concepts and keys to the portal of understanding. *Threshold Concepts within the Disciplines, Sense Publishers*. Retrieved from http://staffwww.itn.liu.se/~jonbe/fou/didaktik/papers/Carstensen_Bernhard_Threshold_draft.pdf

Carter, J., & Jenkins, T. (1999). Gender and programming: What's going on? In *ACM SIGCSE Bulletin* (Vol. 31, pp. 1–4). Retrieved from http://dl.acm.org/citation.cfm?id=305824

Case, J. M. (2008). *Education theories on learning: An informal guide for the engineering education sector*. Retrieved from http://exchange.ac.uk/downloads/scholarart/education-theories.pdf

Case, J. M., & Gunstone, R. (2002). Metacognitive Development as a Shift in Approach to Learning: An in-depth study. *Studies in Higher Education*, *27*(4), 459–470. doi:10.1080/0307507022000011561

Case, J. M., & Light, G. (2011). Emerging methodologies in engineering education research. *Journal of Engineering Education*, *100*(1), 186–210.

Caspi, P., Sangiovanni-Vincentelli, A., Almeida, L., Benveniste, A., Bouyssounouse, B., Buttazzo, G., … Folher, G. (2005). Guidelines for a graduate curriculum on embedded software and systems. *ACM Transactions on Embedded Computing Systems (TECS)*, *4*(3), 587–611.

Catalano, G. D., & Catalano, K. C. (1997). Transformation: from teacher-centered to student-centered engineering education. In *Frontiers in Education Conference, 1997. 27th Annual Conference.'Teaching and Learning in an Era of Change'. Proceedings.* (Vol. 1, pp. 95–100).

References

Chaiklin, S. (2003). The zone of proximal development in Vygotsky's analysis of learning and instruction. *Vygotsky's Educational Theory in Cultural Context*, 39–64.

Chandler, P., & Mayer, R. E. (2001). When learning is just a click away: Does simple user interaction foster deeper understanding of multimedia messages. *Journal of Educational Psychology*, *93*, 390–397.

Chen, T.-Y., Monge, A., & Simon, B. (2006). Relationship of early programming language to novice generated design. In *ACM SIGCSE Bulletin* (Vol. 38, pp. 495–499). Retrieved from http://dl.acm.org/citation.cfm?id=1121496

Cho, S.-Y. (2009). A virtual simulation package for Embedded System training and education. In *Engineering Education (ICEED), 2009 International Conference on* (pp. 72–76). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5490609

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. In S. Fincher & M. Petre (Eds.), *Computer science education research* (pp. 85–100).

Cleary, S. (2010). Various Implementations of Asynchronous Background Tasks. *Stephen Cleary (the blog)*. Retrieved April 11, 2013, from http://blog.stephencleary.com/2010/08/various-implementations-of-asynchronous.html

Coll, R., Taylor, I., & France, B. (2005). The role of models/and analogies in science education: implications from research. *International Journal of Science Education*, *27*(2), 183–198. doi:10.1080/0950069042000276712

Collis, B. (2013). *An introduction to practical electronics, microcontrollers and software design*. Unpublished online textbook, Auckland, New Zealand: Online. Retrieved from http://www.techideas.co.nz/ https://dl.dropboxusercontent.com/u/5571446/IntroToPracticalElectronicsMicrocontrollersAndSoftwareDesign.pdf

Compton, V. J. (2004). Technological Knowledge: A developing framework for technology education in New Zealand. *Ministry of Education, New Zealand*.

References

Compton, V. J. (2007). The role of technology education in supporting a democratic literacy. Retrieved from http://www.tenz.org.nz/2007/Keynote_2.pdf

Compton, V. J., & Compton, A. D. (2011). Teaching the nature of technology: determining and supporting student learning of the philosophy of technology. *International Journal of Technology and Design Education*. doi:10.1007/s10798-011-9176-2

Compton, V. J., & France, B. (2006). Discussion document: background information on the new strands. *Ministry of Education's New Zealand Curriculum Marautanga Project*. Retrieved from http://techlink.org.nz/GIF-tech-education/resources/Discussion-Document-New-Strands-Final.pdf

Compton, V. J., & France, B. (2007). Towards a new technological literacy: Curriculum development with a difference. *Curriculum Matters, NZCER*, (3), 158.

Compton, V. J., & Harwood, C. D. (2003). Enhancing technological practice: An assessment framework for technology education in New Zealand. *International Journal of Technology and Design Education*, *13*(1), 1–26.

Cotterell, S., Vahid, F., Najjar, W., & Hsieh, H. (2003). First results with eBlocks: embedded systems building blocks. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (pp. 168–175). Retrieved from http://dl.acm.org/citation.cfm?id=944690

Coull, N. J., & Duncan, I. M. (2011). Emergent requirements for supporting introductory programming. *ITALICS. 10 (1)*. Retrieved from https://repository.abertay.ac.uk/jspui/handle/10373/710

Crenshaw, J. W. (1988). *Let's Build a Compiler!* Disponible en URL:< http://www. maththinking. com/boat/booksIndex. html.>.[Acceso en marzo de 2007]. Retrieved from http://www.penguin.cz/~radek/book/lets_build_a_compiler.pdf

Crews, T. (2001). Using a Flowchart Simulator in an Introductory Programming Course. *Computer Science Teaching Centre Digital Library, Western Kentucky University, USA. Http://www. Citidel. org/bitstream/10117/119/2/Visual. Pdf*.

References

Custer, R. L. (1995). Examining the dimensions of technology. *International Journal of Technology and Design Education*, *5*(3), 219–244.

De Raadt, M. (2008a). Strategies Reference. Retrieved from http://dl.dropboxusercontent.com/u/11561272/research/dissertation/Strategies%20Reference.pdf

De Raadt, M. (2008b). *Teaching programming strategies explicitly to novice programmers*. Australia: Unpublished doctoral dissertation, University of Southern Queensland. Retrieved from http://eprints.usq.edu.au/4827/2/de_Raadt_2008_whole.pdf

De Raadt, M., Watson, R., & Toleman, M. (2009a). Teaching and assessing programming strategies explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95* (pp. 45–54). Retrieved from http://dl.acm.org/citation.cfm?id=1862723

De Raadt, M., Watson, R., & Toleman, M. (2009b). Teaching and assessing programming strategies explicitly. In *Proceedings of the 11th Australasian Computing Education Conference (ACE 2009)* (Vol. 95, pp. 45–54). Retrieved from http://eprints.usq.edu.au/4879

Decoo, W. (1996). The induction-deduction opposition: Ambiguities and complexities of the didactic reality. *Iral*, *34*(2), 95–118.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 208–212). Retrieved from http://dl.acm.org/citation.cfm?id=1999807

Doboli, A., Doboli, S., & Currie, E. H. (2008). Visual embedded system programming has arrived! In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual* (p. F3E–9). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4720577

Dreyfus, S. E., & Dreyfus, H. L. (1980). *A five-stage model of the mental activities involved in directed skill acquisition*. DTIC Document.

References

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*(3), 237–249.

Ebenezer, J. V., & Fraser, D. M. (2001). First year chemical engineering students' conceptions of energy in solution processes: Phenomenographic categories for common knowledge construction. *Science Education*, *85*(5), 509–535.

EBNF Visualizer. (2013). *Sourceforge*. Retrieved April 20, 2013, from http://ebnfvisualizer.sourceforge.net/

Eckerdal, A., & Berglund, A. (2005). What does it take to learn'programming thinking'? In *Proceedings of the first international workshop on Computing education research* (pp. 135–142). Retrieved from http://dl.acm.org/citation.cfm?id=1089799

Edwards, S. (1998). *Programming and Customizing the Basic Stamp Computer*. McGraw-Hill, Inc.

Eggen, P. D., & Kauchak, D. P. (2006). *Strategies and models for teachers* (5th Edition.). Pearson Publishing.

Engel, G., & Roberts, E. (Eds.). (2001). Computer Science, Final Report, The Joint Task Force on Computing Curricula. *IEEE Computer Society and Association for Computing Machinery, IEEE Computer Society*.

Ernst, E. W. (1983). A new role for the undergraduate engineering laboratory. *Education, IEEE Transactions on*, *26*(2), 49–51.

Ertmer, P. A., & Newby, T. J. (2013). Behaviorism, Cognitivism, Constructivism: Comparing Critical Features From an Instructional Design Perspective. *Performance Improvement Quarterly*, *26*(2), 43–71.

Feenberg, A. (2010). *Between reason and experience: Essays in technology and modernity*. The MIT Press.

References

Felder, R. M. (2012). Engineering Education–A Tale of Two Paradigms. In *SFGE, 2nd. Int Conf on Geotechnical Engineering Education, Galway*. Retrieved from http://www4.ncsu.edu/unity/lockers/users/f/felder/public/Papers/TwoParadigms.pdf

Felder, R. M., & Brent, R. (2004a). The intellectual development of science and engineering students. Part 1: Models and challenges. *Journal of Engineering Education*, *93*(4), 269–278.

Felder, R. M., & Brent, R. (2004b). The intellectual development of science and engineering students. Part 2: Teaching to Promote Growth. *Journal of Engineering Education*, *93*(4), 279–291.

Felder, R. M., & Silverman, L. K. (1988). Learning and teaching styles in engineering education. *Engineering Education*, *78*(7), 674–681.

Fincher, S., & Petre, M. (2004). *Computer science education research*. Psychology Press.

Fischer, C. N., Cytron, R. K., & LeBlanc, R. J. (2009). *Crafting a compiler*. Addison-Wesley Publishing Company.

Fischer, R. A. (1979). The Inductive-Deductive Controversy Revisited. *The Modern Language Journal*, *63*(3), 98–105.

Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual* (p. T3H–10). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1408551

France, B., Compton, V. J., & Gilbert, J. K. (2010). Understanding modelling in technology and science: the potential of stories from the field. *International Journal of Technology and Design Education*, *21*(3), 381–394. doi:10.1007/s10798-010-9126-4

Freiermuth, K., Hromkovič, J., & Steffen, B. (2008). Creating and testing textbooks for secondary schools. *Informatics Education-Supporting Computational Thinking*, 216–228.

Freire, P. (2004). *EPZ Pedagogy of Hope: Reliving Pedagogy of the Oppressed*. Continuum.

Gartner Research. (2013). Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013. *Gartner*. Retrieved April 11, 2013, from http://www.gartner.com/newsroom/id/2408515

References

Gemino, A., & Wand, Y. (2004). A framework for empirical evaluation of conceptual modeling techniques. *Requirements Engineering*, *9*(4), 248–260.

Getty, J. C. (2009). Assessing inquiry learning in a circuits/electronics course. In *Frontiers in Education Conference, 2009. FIE'09. 39th IEEE* (pp. 1–6). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5350849

Gilbert, J. K. (2004). Models and modelling: Routes to more authentic science education. *International Journal of Science and Mathematics Education*, *2*(2), 115–130.

Giovani, B. (2011). 10 Billionth Microchip PIC Microcontroller Shipped. *Microcontroller.com*. Retrieved February 12, 2012, from a://microcontroller.com/news/10_billionth_microchip_pic_microcontroller.asp

Glaser, R. (1984). Education and thinking: The role of knowledge. *American Psychologist*, *39*(2), 93.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, *40*(1), 256–260.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Transactions on Computing Education*, *10*(2), 1–29. doi:10.1145/1789934.1789935

Gomes, A., & Mendes, A. J. (2007). Learning to program-difficulties and solutions. In *International Conference on Engineering Education–ICEE* (Vol. 2007). Retrieved from http://ineer.org/Events/ICEE2007/papers/411.pdf

Goris, T., & Dyrenfurth, M. (2010). Students' Misconceptions in Science, Technology, and Engineering. Retrieved from http://ilin.asee.org/Conference2010/Papers/A1_Goris_Fyrenfurth.pdf

Gott, S. P. (1988). Apprenticeship instruction for real-world tasks: The coordination of procedures, mental models, and strategies. *Review of Research in Education*, *15*, 97–169.

References

Greco, J. F., & Nestor, J. A. (2011). An undergraduate embedded systems project. In *Microelectronic Systems Education (MSE), 2011 IEEE International Conference on* (pp. 43–46). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5937089

Grimheden, M., & Törngren, M. (2005). What is embedded systems and how should it be taught?—results from a didactic analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, *4*(3), 633–651.

Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research*, *2004*, 127–154.

Guzdial, M. (2010). Why is it so hard to learn to program? In *Making Software: What Really Works, and Why We Believe It* (pp. 111–124). O'Reilly Media, Incorporated.

Hanke, U. (2008). Realizing Model-Based Instruction. In *Understanding models for learning and instruction* (pp. 175–186). Springer. Retrieved from http://link.springer.com/chapter/10.1007/978-0-387-76898-4_9

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231–274.

Harel, D. (2009). Statecharts in the making: a personal account. *Communications of the ACM*, *52*(3), 67–75.

Harel, D., & Gordon-Kiwkowitz, M. (2009). On teaching visual formalisms. *Software, IEEE*, *26*(3), 87–95.

Harkin, J., Callaghan, M. J., McGinnity, T. M., & Maguire, L. P. (2002). An Internet based remote access experimental laboratory for embedded systems. In *Engineering Education 2002: Professional Engineering Scenarios (Ref. No. 2002/056), IEE* (Vol. 1, pp. 18–1). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1028460

Harwood, C. D. (2002). Technology in New Zealand Education Conference. Presented at the Graphics and Technology Teachers Association, Christchurch.

References

Harwood, C. D., & Compton, V. J. (2007). Moving from technical to technology education: Why it's so hard. In *TENZ Biennial Conference* (Vol. 7). Retrieved from http://www.tenz.org.nz/2007/Tech_Paper9.pdf

Hattie, J. (2011). Visible Learning For Teachers: Maximizing Impact On Learning Author: John Hattie, Publisher: Routledge Pages: 280 Publish.

Hattie, J., & Jaeger, R. (1998). Assessment and Classroom Learning: a deductive approach. *Assessment in Education: Principles, Policy & Practice*, *5*(1), 111–122. doi:10.1080/0969595980050107

Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of Educational Research*, *77*(1), 81–112.

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259–290.

Janka, R. S. (2002). *Specification and design methodology for real-time embedded systems*. Springer.

Johnson-Laird, P. (2006). Mental models, sentential reasoning, and illusory inferences. *Advances in Psychology*, *138*, 27–51.

Johri, A., & Olds, B. M. (2011). Situated engineering learning: Bridging engineering education research and the learning sciences. *Journal of Engineering Education*, *100*(1), 151–185.

Jonassen, D. H. (1995). Operationalizing mental models: strategies for assessing mental models to support meaningful learning and design-supportive learning environments. In *The first international conference on Computer support for collaborative learning* (pp. 182–186). Retrieved from http://ite.stu.edu.cn/xdjyjs/lunwen/paper/s2/74.htm

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 107–111). Retrieved from http://dl.acm.org/citation.cfm?id=1734299

Kaila, E., Rajala, T., Laakso, M., & Salakoski, T. (2011). Important features in program visualization. In *International Conference on Engineering Education–ICEE* (pp. 21–26).

Kalyuga, S. (2010). Schema acquisition and sources of cognitive load. In *Cognitive Load Theory* (pp. 48–64). Cambridge University Press.

References

Kaplan, B., & Maxwell, J. A. (2005). Qualitative research methods for evaluating computer information systems. *Evaluating the Organizational Impact of Healthcare Information Systems*, 30–55.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), 83–137.

Kelly, A., & Pohl, I. (1990). *A book on C: programming in C* (2nd ed.). Benjamin-Cummings Publishing Co.

Kern, C., & Greenstreet, M. R. (1999). Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *4*(2), 123–193.

Kester, L., Paas, F. G., & Van Merriënboer, J. J. (2010). Instructional control of cognitive load in the design of complex environments. In *Cognitive Load Theory* (pp. 109–130). Cambridge University Press.

Kinnunen, P., & Simon, B. (2012). Phenomenography and grounded theory as research methods in computing education research field. *Computer Science Education*, *22*(2), 199–218. doi:10.1080/08993408.2012.692928

Kirschner, P. A. (2002). Cognitive load theory: Implications of cognitive load theory on the design of learning. *Learning and Instruction*, *12*(1), 1–10.

Koopman, P., Choset, H., Gandhi, R., Krogh, B., Marculescu, D., Narasimhan, P., … Smailagic, A. (2005). Undergraduate embedded system education at Carnegie Mellon. *ACM Transactions on Embedded Computing Systems (TECS)*, *4*(3), 500–528.

Kossiakoff, A., Sweet, W. N., Seymour, S. J., & Biemer, S. M. (2011). *Systems engineering principles and practice*. Wiley Online Library.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into Practice*, *41*(4), 212–218.

Krone, J., Hollingsworth, J. E., Sitaraman, M., & Hallstrom, J. O. (2010). *A reasoning concept inventory for computer science*. unpublished technical report RSRG-01-01, School of Computing, Clemson. Retrieved from http://www.cs.clemson.edu/resolve/research/reports/RSRG-10-01.pdf

References

Kuittinen, M., & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. In *ACM SIGCSE Bulletin* (Vol. 36, pp. 57–61).

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research*, *2*(4), 429–458.

Laakso, M. (2013). ViLLE - the visual learning tool. *ViLLE*. Retrieved March 12, 2013, from http://ville.cs.utu.fi/?pg=6

Laakso, M., Kaila, E., & Salakoski, T. (2008). Effectiveness of Program Visualization: A Case Study with the ViLLE Tool. *Journal of Information Technology Education*, *7*. Retrieved from http://jite.informingscience.org/documents/Vol7/JITEv7IIP015-032Rajala394.pdf

Lahtinen, E., Ahoniemi, T., & Salo, A. (2007). Effectiveness of integrating program visualizations to a programming course. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88* (pp. 195–198). Retrieved from http://dl.acm.org/citation.cfm?id=2449350

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin* (Vol. 37, pp. 14–18).

Lahtinen, E., Järvinen, H.-M., & Melakoski-Vistbacka, S. (2007). Targeting program visualizations. *ACM SIGCSE Bulletin*, *39*(3), 256–260.

Land, R., Cousin, G., Meyer, J. H. F., & Davies, P. (2005). Threshold concepts and troublesome knowledge (3): implications for course design and evaluation. *Improving Student Learning–equality and Diversity, Oxford: OCSLD*. Retrieved from http://161.73.1.13/services/ocsld/isl/isl2004/abstracts/conceptual_papers/ISL04-pp53-64-Land-et-al.pdf

Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation*. Cambridge university press.

References

Laverty, D. M., Milliken, J., Milford, M., & Cregan, M. (2012). Embedded C programming: a practical course introducing programmable microprocessors. *European Journal of Engineering Education*, 1–18. doi:10.1080/03043797.2012.725711

Lee, E., & Seshia, S. (2011). *Introduction to embedded systems: a cyber-physical systems approach*. LeeSeshia.org.

Letovsky, S., & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, *3*(3), 41–49.

Levy, R. B.-B., & Ben-Ari, M. (2007). We work so hard and they don't use it: acceptance of software tools by teachers. *ACM SIGCSE Bulletin*, *39*(3), 246–250.

Levy, R. B.-B., & Ben-Ari, M. (2008). Perceived behavior control and its influence on the adoption of software tools. In *ACM SIGCSE Bulletin* (Vol. 40, pp. 169–173). Retrieved from http://dl.acm.org/citation.cfm?id=1384318

Leydens, J. A., Moskal, B. M., & Pavelich, M. J. (2004). Qualitative methods used in the assessment of engineering education. *Journal of Engineering Education*, *93*(1), 65–72.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction in classrooms. *Educational Researcher*, *14*(5), 14–29.

Liu, C. H., & Matthews, R. (2005). Vygotsky's philosophy: Constructivism and its criticisms examined. *International Education Journal*, *6*(3), 386–399.

Lopes, L. (2013). PICsim. *PICsim - PIC microcontroller simulator*. Retrieved March 12, 2013, from http://sourceforge.net/projects/picsim/

Ma, L., Ferguson, J. D., Roper, M., Ross, I., & Wood, M. (2008). Using cognitive conflict and visualisation to improve mental models held by novice programmers. In *ACM SIGCSE Bulletin* (Vol. 40, pp. 342–346). Retrieved from http://dl.acm.org/citation.cfm?id=1352253

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 499–503). Retrieved from http://dl.acm.org/citation.cfm?id=1227310.1227481

References

Magrini, J. (2009). How the conception of knowledge influences our educational practices: Toward a philosophical understanding of epistemology in education. *College of DuPage, Digital Commons*, *Curriculum Matters*(6). Retrieved from http://dc.cod.edu/philosophypub/13/

Mak, R. L. (1996). *Writing Compilers and Interpreters: An Applied Approach Using C++*. John Wiley & Sons, Inc.

Malan, K., & Halland, K. (2004). Examples that can do harm in learning programming. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 83–87). Retrieved from http://dl.acm.org/citation.cfm?id=1028702

Mantri, A., Dutt, S., Gupta, J. P., & Chitkara, M. (2009). Using PBL to deliver course in Digital Electronics. *Advances in Engineering Education, Spring*. Retrieved from http://advances.asee.org/vol01/issue04/papers/aee-vol01-issue04-p06.pdf?q=jaypee-institute-of-information-technology

Manus, D. A. L. (1996). Procedural versus constructivist education: A lesson from history. In *The Educational Forum* (Vol. 60, pp. 312–316).

Marshall, J. (1988). *Why Go to School?* Dunmore Press.

Marton, F. (1981). Phenomenography—describing conceptions of the world around us. *Instructional Science*, *10*(2), 177–200.

Marton, F., Tsui, A. B., & Runesson, U. (2004). The space of learning. In *Classroom discourse and the space of learning* (pp. 3–40). Lawrence Erlbaum Associates.

Mason, R., Cooper, G., & de Raadt, M. (2012). Trends in Introductory Programming Courses in Australian Universities–Languages, Environments and Pedagogy. *Reproduction*, 33–42.

Mawson, B. (1998). Technology: flogging a dead horse or beating the odds. *ACE Papers*, *2*, 38–50.

Mayer, R. E. (1987). Cognitive aspects of learning and using a programming language. In J. M. Carroll (Ed.), *Interfacing thought: Cognitive aspects of humna-computer interation* (pp. 61–79). The MIT Press.

References

Mayer, R. E. (1989). Models for Understanding. *Review of Educational Research*, *59*(1), 43–64. doi:10.3102/00346543059001043

Mayer, R. E. (1992). Cognition and instruction: Their historic meeting within educational psychology. *Journal of Educational Psychology*, *84*(4), 405.

Mayer, R. E., & Moreno, R. (2003). Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist*, *38*(1), 43–52.

Mayer, R. E., & Moreno, R. (2010). Techniques that reduce extraneous cognitive load and manage intrinsic cognitive load during multimedia learning. In *Cognitive Load Theory* (pp. 131–152). Cambridge University Press.

McCartney, R., Boustedt, J., Eckerdal, A., Moström, J. E., Sanders, K., Thomas, L., & Zander, C. (2009). Liminal spaces and learning computing. *European Journal of Engineering Education*, *34*(4), 383–391. doi:10.1080/03043790902989580

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K., & Zander, C. (2013). Can first-year students program yet?: a study revisited. In *Proceedings of the ninth annual international ACM conference on International computing education research* (pp. 91–98).

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92.

McCormick, R. (1997). Conceptual and procedural knowledge. *International Journal of Technology and Design Education*, *7*(1), 141–159.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., … Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125–180.

MCS Electronics. (2013). MCS Electronics. *MCS Electronics*. Retrieved March 12, 2013, from http://mcselec.com/

References

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *ACM SIGCSE Bulletin*, *21*(4), 182–194.

Medin, D. L., Rips, L. J., & Smith, E. (2012). Concepts and categories: Memory, meaning, and metaphysics. *The Oxford Handbook of Thinking and Reasoning*, 37–72. doi:10.1093/oxfordhb/9780199734689.001.0001

Microsoft Corporation. (2013a). BackgroundWorker Component Overview. *MSDN*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/8xs8549b.aspx

Microsoft Corporation. (2013b). Dictionary<TKey, TValue>.Add Method. *MSDN*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/k7z0zy8k.aspx

Microsoft Corporation. (2013c). How to: Make Thread-Safe Calls to Windows Forms Controls. *MSDN*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/ms171728.aspx

Microsoft Corporation. (2013d). How to: Run an Operation in the Background. *MSDN*. Retrieved April 7, 2013, from http://msdn.microsoft.com/en-us/library/hybbz6ke.aspx

Microsoft Corporation. (2013e). Managed Threading. *Microsoft Help Library*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/3e8s7xdd.aspx

Microsoft Corporation. (2013f). Managed Threading Best Practices. *Microsoft Help Library*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/1c9txz50.aspx

Microsoft Corporation. (2013g). Wait Handles. *MSDN*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/kad9xah9.aspx

Microsoft Corporation. (2013h). Walkthrough: Multithreading with the BackgroundWorker Component (C# and Visual Basic)MethodInvoker Delegate. *MSDN*. Retrieved April 11, 2013, from http://msdn.microsoft.com/en-us/library/vstudio/ywkkz4s1.aspx

Ming, L., Longxing, S., & Yongming, T. (2012). An Advanced C Programming Course for the Embedded Systems Development. In *IEEE International Conference on Teaching, Assessment, and Learning for Engineering*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6360328

References

Ministry of Education. (2007). The New Zealand Curriculum. Retrieved from http://nzcurriculum.tki.org.nz/Curriculum-documents

Ministry of Education. (2010). Technology curriculum support. *Technology Online*. Retrieved March 9, 2013, from http://technology.tki.org.nz/Curriculum-support

Ministry of Education. (2013). The school curriculum: Design and review. Retrieved April 20, 2013, from http://nzcurriculum.tki.org.nz/Curriculum-documents/The-New-Zealand-Curriculum/The-school-curriculum-Design-and-review

Mioduser, D., & Betzer, N. (2007). The contribution of Project-based-learning to high-achievers' acquisition of technological knowledge and skills. *International Journal of Technology and Design Education*, *18*(1), 59–77. doi:10.1007/s10798-006-9010-4

Mohan, S., & Chenoweth, S. (2011). Teaching requirements engineering to undergraduate students. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 141–146). Retrieved from http://dl.acm.org/citation.cfm?id=1953207

Moreno, R., & Park, B. (2010). Cognitive load theory: Historical development and relation to other theories. In *Cognitive Load Theory* (pp. 9–28). Cambridge University Press.

Mosemann, R., & Wiedenbeck, S. (2001). Navigation and comprehension of programs by novice programmers. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on* (pp. 79–88). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=921716

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin* (Vol. 40, pp. 163–167). Retrieved from http://dl.acm.org/citation.cfm?id=1352191

Myers, M. D. (1997). Qualitative research in information systems. *Management Information Systems Quarterly*, *21*, 241–242.

References

Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rößling, G., Dann, W., … others. (2003). Evaluating the educational impact of visualization. In *ACM SIGCSE Bulletin* (Vol. 35, pp. 124–136). Retrieved from http://dl.acm.org/citation.cfm?id=960540

Nevalainen, S., & Sajaniemi, J. (2006). An experiment on short-term effects of animated versus static visualization of operations on program perception. In *Proceedings of the second international workshop on Computing education research* (pp. 7–16). Retrieved from http://dl.acm.org/citation.cfm?id=1151591

Packer, M. J., & Goicoechea, J. (2000). Sociocultural and constructivist theories of learning: Ontology, not just epistemology. *Educational Psychologist*, *35*(4), 227–241.

Parsons, D., & Haden, P. (2007). Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Conference of the National Advisory Committee on Computing Qualifications. Citeseer*. Retrieved from http://www.citrenz.ac.nz/conferences/2007/209.pdf

Paulk, M. C., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993). Capability maturity model, version 1.1. *Software, IEEE*, *10*(4), 18–27.

Pea, R. D. (1986). Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36.

Perkins, D. N. (1991). Educating for insight. *Educational Leadership*, *49*(2), 4–8.

Perkins, D. N. (1999). The many faces of constructivism. *Educational Leadership*, *57*(3), 6–11.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, *2*(1), 37–55.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In R. E. Mayer (Ed.), *Teaching and learning computer programming* (pp. 153–178).

Prince, M. J. (2004). Does active learning work? A review of the research. *Journal of Engineering Education*, *93*, 223–232.

Prince, M. J., & Felder, R. M. (2006). Inductive teaching and learning methods: Definitions, comparisons, and research bases. *Journal of Engineering Education*, *95*(2), 123.

References

Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A Summary of Misconceptions of High School Basic Programmers. *Journal of Educational Computing Research*, *2*(4), 459–472. doi:10.2190/FGN9-DJ2F-86V8-3FAU

Rajala, T., Salakoski, T., Laakso, M., & Kaila, E. (2009). Effects, experiences and feedback from studies of a program visualization tool. *Informatics in Education-An International Journal*, (Vol 8_1), 17–34.

Ramsden, P., Masters, G. N., Stephanou, A., Walsh, E., Martin, E., Laurillard, D., & Marton, F. (1993). Phenomenographic research and the measurement of understanding: an investigation of students' conceptions of speed, distance, and time.

Revolution Education Ltd. (2013). PICAXE VSM. Retrieved April 11, 2013, from http://www.picaxe.com/Software/PICAXE/PICAXE-VSM/

Richardson, J. T. E. (1999). The Concepts and Methods of Phenomenographic Research. *Review of Educational Research*, *69*(1), 53–82. doi:10.3102/00346543069001053

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*(3), 389–414.

Robal, T., Kann, T., & Kalja, A. (2011). An ontology-based intelligent learning object for teaching the basics of digital logic. In *Microelectronic Systems Education (MSE), 2011 IEEE International Conference on* (pp. 106–107). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5937105

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Rogalski, J., & Samurcay, R. (1990). Acquisition of programming knowledge and skills. In *Psychology of programming* (pp. 157–174). Academic Press.

Ropohl, G. (1997). Knowledge types in technology. *International Journal of Technology and Design Education*, *7*(1), 65–72.

Rountree, J., Rountree, N., Robins, A., & Hannah, R. (2005). Observations of student competency in a CS1 course. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42* (pp. 145–149). Retrieved from http://dl.acm.org/citation.cfm?id=1082442

Rowe, G. B., & Smaill, C. (2007). Work in progress-A web-based system for the delivery and analysis of course concept inventories. In *Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE'07. 37th Annual* (p. F2H–24). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4417920

Sajaniemi, J. (2002). Visualizing roles of variables to novice programmers. In *Psychology of Programming workshop PPIG 2002* (pp. 111–127).

Sajaniemi, J., & Kuittinen, M. (2003). Program animation based on the roles of variables. In *Proceedings of the 2003 ACM symposium on Software visualization* (pp. 7–16).

Samurcay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. *Studying the Novice Programmer*, *9*, 161–178.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L., & Zander, C. (2012). Threshold concepts and threshold skills in computing. In *Proceedings of the ninth annual international conference on International computing education research* (pp. 23–30). Retrieved from http://dl.acm.org/citation.cfm?id=2361283

Saraiya, P., Shaffer, C. A., McCrickard, D. S., & North, C. (2004). *Effective features of algorithm visualizations* (Vol. 36). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=971432

Schulte, C. (2008). Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceeding of the fourth international workshop on Computing education research* (pp. 149–160). Retrieved from http://dl.acm.org/citation.cfm?id=1404535

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? In *Proceedings of the second international workshop on Computing education research* (pp. 17–28). Retrieved from http://dl.acm.org/citation.cfm?id=1151593

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports* (pp. 65–86). Retrieved from http://dl.acm.org/citation.cfm?id=1971687

References

Scott, A., Watkins, M., & McPhee, D. (2008). E-Learning For Novice Programmers. Retrieved from http://www.comp.glam.ac.uk/staff/asscott/papers/ICTTA-08.doc

Sfard, A. (1998). On two metaphors for learning and the dangers of choosing just one. *Educational Researcher*, *27*(2), 4–13.

Shinners-Kennedy, D., & Fincher, S. A. (2013). Identifying threshold concepts: from dead end to a new direction (p. 9). ACM Press. doi:10.1145/2493394.2493396

Shulman, L. S. (1986). Those who understand: Knowledge growth in teaching. *Educational Researcher*, 4–14.

Shulman, L. S. (1987). Knowledge and teaching: Foundations of the new reform. *Harvard Educational Review*, *57*(1), 1–23.

Simon, B., Bouvier, D., Chen, T. Y., Lewandowski, G., McCartney, R., & Sanders, K. (2008). Common sense computing (episode 4): Debugging. *Computer Science Education*, *18*(2), 117–133.

Simon, B., Fitzgerald, S., McCauley, R., Haller, S., Hamer, J., Hanks, B., … Thomas, L. (2007). Debugging assistance for novices: a video repository. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 137–151). Retrieved from http://dl.acm.org/citation.cfm?id=1345437

Sinha, R., Roop, P., & Basu, S. (2014). Automatic Verification Using Model and Module Checking. In *Correct-by-Construction Approaches for SoC Design* (pp. 25–54). Springer.

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, *2*(1), 5–23. doi:10.2190/2XPP-LTYH-98NQ-BU77

Snow, A. (2013). ProgrAnimate. *ProgrAnimate*. Retrieved March 12, 2013, from http://www.prograniMate.com/

Soft Integration. (2013). “Ch” interpreter for C. *SoftIntegration*. Retrieved April 10, 2013, from http://www.softintegration.com/

References

Soloway, E. (1985). From Problems to Programs Via Plans: The Content and Structure of Knowledge for Introductory LISP Programming. *Journal of Educational Computing Research*, *1*(2), 157–172. doi:10.2190/WK8C-BYCF-VQ5C-E307

Soloway, E. (1986). Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. doi:10.1145/2483710.2483713

Sorva, J., Karavirta, V., & Korhonen, A. (2007). Roles of variables in teaching. *Journal of Information Technology Education*, *6*(2007), 407–423.

Spohrer, J. C., & Soloway, E. (1986). Alternatives to construct-based programming misconceptions. In *ACM SIGCHI Bulletin* (Vol. 17, pp. 183–191). Retrieved from http://dl.acm.org/citation.cfm?id=22369

Stern, L., Markham, S., & Hanewald, R. (2005). You can lead a horse to water: how students really use pedagogical software. *ACM SIGCSE Bulletin*, *37*(3), 246–250.

Sternberg, R. J. (2010). *Educational psychology / Robert J. Sternberg, Wendy M. Williams.* Upper Saddle River, N.J. : Merrill, c2010.

Sudol-DeLyser, L. A., Stehlik, M., & Carver, S. (2012). Code comprehension problems as learning events. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (pp. 81–86). Retrieved from http://dl.acm.org/citation.cfm?id=2325319

Tew, A. E., Dorn, B., & Schneider, O. (2012). Toward a validated computing attitudes survey. In *Proceedings of the ninth annual international conference on International computing education research* (pp. 135–142). Retrieved from http://dl.acm.org/citation.cfm?id=2361303

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 97–101). Retrieved from http://dl.acm.org/citation.cfm?id=1734297

References

Tew, A. E., & Guzdial, M. (2011). The FCS1: a language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 111–116). Retrieved from http://dl.acm.org/citation.cfm?id=1953200

The Databeans Monthly. (2012). Microcontroller Market Share: In 3 Dimensions. *Databeans.Net*. Retrieved April 11, 2013, from http://www.databeans.net/newsletter/pdfs_nl/March-2012.pdf

Thornburg, D. (1994). Killing the fatted calf: Skinner recanted behaviorism. Why can't education? *Electronic Learning*, *14*(1), 24–25.

Thuné, M., & Eckerdal, A. (2009). Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education*, *34*(4), 339–347. doi:10.1080/03043790902989374

Tomlinson, C. A. (2008). *The differentiated school: Making revolutionary changes in teaching and learning*. Alexandria, VA: Associated for Supervision and Curriculum Development.

Tomlinson, C. A., & Allan, S. D. (2006). *Leadership for Differentiating Schools and Classrooms*. Heatherton, Victoria: Hawker Brownlow Education.

Tomlinson, C. A., Brighton, C. M., Hertberg, H. L., Callahan, C. M., Moon, T. R., Brimijoin, L. A., & Reynolds, T. (2003). Differentiated Instruction in response to student readiness, interest and learning profile in academically diverse classrooms: A review of literature. *Journal for the Education of the Gifted*, *27*(2), 119–145.

Torczon, L., & Cooper, K. (2007). *Engineering A Compiler*. Morgan Kaufmann Publishers Inc.

Torgashov, P. (2013). Fast Colored TextBox for Syntax Highlighting. *Code Project*. Retrieved March 12, 2013, from http://www.codeproject.com/Articles/161871/Fast-Colored-TextBox-for-syntax-highlighting

Tseng, K.-H., Chang, C.-C., Lou, S.-J., & Chen, W.-P. (2011). Attitudes towards science, technology, engineering and mathematics (STEM) in a project-based learning (PjBL) environment. *International Journal of Technology and Design Education*. doi:10.1007/s10798-011-9160-x

References

Tuma, T., & Fajfar, I. (2006). A new curriculum for teaching embedded systems at the University of Ljubljana. In *Information Technology Based Higher Education and Training, 2006. ITHET'06. 7th International Conference on* (pp. 14–19). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4141601

UBM Tech. (2013). *2013 Embedded Market Study Results*. Retrieved from http://e.ubmelectronics.com/2013EmbeddedStudy/index.html

University of California. (2013). eBlocks : Embedded Systems Building Blocks. *eBlocks*. Retrieved March 12, 2013, from http://www.cs.ucr.edu/~eblock/

Urquiza-Fuentes, J., & Velázquez-Iturbide, J. Á. (2009). A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Transactions on Computing Education (TOCE)*, *9*(2), 9.

Vahid, F., & Givargis, T. (2002). *Embedded system design: a unified hardware/software introduction* (Vol. 4). John Wiley & Sons New York, NY.

Van Merriënboer, J. J., & Paas, F. G. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior*, *6*(3), 273–289.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494.

Vygotsky, L. S. (1987). *The collected works of LS Vygotsky: Volume 1: Problems of general psychology, including the volume Thinking and Speech* (Vol. 1). Springer.

Warschauer, M. (2004). *Technology and social inclusion : rethinking the digital divide*. The MIT Press.

WestAust55. (2011). PICAXE Program size Optimisation and Speed. Retrieved February 12, 2012, from http://www.picaxeforum.co.uk/showthread.php?17782-PICAXE-program-Size-Optimisations-and-Speed

References

Williams, P. J., Iglesias, J., & Barak, M. (2007). Problem based learning: application to technology education in three countries. *International Journal of Technology and Design Education*, *18*(4), 319–335. doi:10.1007/s10798-007-9028-2

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.

Winzker, M., & Schwandt, A. (2011). Teaching Embedded System Concepts for Technological Literacy. *Education, IEEE Transactions on*, *54*(2), 210–215.

Wolf, W. H. (1994). Hardware-software co-design of embedded systems [and prolog]. *Proceedings of the IEEE*, *82*(7), 967–989.

Worster, W. (2013). The Inductive and Deductive Methods in Customary International Law Analysis: Traditional and Modern Approaches. *Available at SSRN 2197104*. Retrieved from http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2197104

Yadav, A., Subedi, D., Lundeberg, M. A., & Bunting, C. F. (2011). Problem-based learning: Influence on students' learning in an electrical engineering course. *Journal of Engineering Education*, *100*(2), 253–280.

Yamamoto, M., Sekiya, T., & Yamaguchi, K. (2011). Relationship between programming concepts underlying programming skills. In *Information Technology Based Higher Education and Training (ITHET), 2011 International Conference on* (pp. 1–7). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6018678

Zendler, A., & Spannagel, C. (2008). Empirical foundation of central concepts for computer science education. *Journal on Educational Resources in Computing (JERIC)*, *8*(2), 6.

Glossary

Algorithm visualisation: stylised and animated view of the flow of data in an algorithm

AVR: A range of 8 bit microcontrollers manufactured by Atmel Corporation, includes the ATMega and ATTiny devices.

Bootloader: this is a short program that runs in the computer when it has power applied. When part of a microcontroller development system, this program runs for only a short period of time before the main program runs so as to allow a user to go into a reprogramming mode rather than the run the main program in memory.

NCEA: National Certificate of Educational Achievement; New Zealand's secondary school qualification system.

NZQA: New Zealand Qualification Authority. Government organisation, one of its roles is to manage NCEA.

Program visualisation: animated view of program execution

Software visualisation: program and algorithm visualisation

WinForms: Windows forms is the graphical application programming interface that is included with Microsoft .NET Framework.

WPF: Windows Presentation Foundation is a subsystem of Microsoft .NET Framework with advanced user interface features.

Appendix A: Participant Information Sheet

Department of Electrical and
Computer Engineering

Science Centre (Building 303)
38 Princes Street
Auckland, New Zealand
Telephone 64 9 373 7599 ext. 88158
Facsimile 64 9 373 7461
www.auckland.ac.nz

The University of Auckland
Private Bag 92019
Auckland 1142

Participant Information Sheet

Project Title: The development of a visualiser application for novice students of embedded systems.

Participation in this research is entirely voluntary

Researcher: Bill Collis, email: wcol001@aucklanduni.ac.nz, billcollis@gmail.com

Supervisor: Dr Gerard Rowe +64 373-7599 extn. 82009, gb.rowe@auckland.ac.nz

Co-Supervisor: Dr Partha Roop, +64 9 373 7599 extn. 85583, p.roop@auckland.ac.nz

Researcher introduction

My name is Bill Collis, and I am a student at the University of Auckland undertaking the Degree of Masters of Engineering in the Department of Electrical and Computer Engineering with the Embedded Systems Research Group. I am conducting research on visualising software and hardware processes for novice embedded systems developers.

Project description

In this study you will be asked to test, debug and write snippets of embedded systems computer programs using a computer based visualiser application. You will be asked to talk about what you are doing and what you are thinking about while you are undertaking the tasks.

Before you begin the task, a demonstration and explanations will be given and code examples for the environment will be available. A voice recording will be made during the tasks you undertake. You will

answer a pre-study and post-study questionnaire which will be used to measure your experience in programming and the visualiser application.

You are invited to participate in this research. We will be delighted if you choose to take part in our research. Participation in the study would require approximately 90 minutes of uninterrupted time. Taking part in the study enables you to contribute to how novice learners begin to understand embedded systems and write programs for them. We hope that this research will uncover ways in which programming for new learners can be improved and lead to their better understanding of embedded systems.

Confidentiality:

The questionnaires, tasks and audio transcripts will not ask for any information that will identify you as the participant. Your results will be completely confidential and will not be made available to staff or other students. Any data collected will have no effect on any of your course grades. The audio recordings will be transcribed by another person; they will not be able to identify you and will not know your name. You will be able to review and edit the transcript if you wish to.

If the information you provide is published, this will be done in a way that does not identify you as its source. The original questionnaires, computer logs, and any electronic files you produced will be stored securely on university premises for six years after the study is completed, at which point they will be destroyed. You may request a copy of your data by contacting the researcher or the supervisors once the user study is complete.

Your participation in this research is voluntary. You will also be able to withdraw your data at any time up to two weeks from the date of your participation. A $20 gift voucher will be given to you as a token of appreciation for your effort to take part in the study.

Please contact the researcher if you want to hear or read about the results of our study and the researcher will be delighted to share them with you. The researcher will seek to publish the study results in research journals, and share the learning with the rest of the research community.

Drs Gerard Rowe and Partha Roop have given assurance that your participation or non-participation will not affect your grades or relationships within the University. You can contact the head of department, Professor Zoran Salcic (Room 303.244, ext. 87802), if you need to make a complaint about this assurance not being upheld.

Chair contact details: —For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Research Office, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 87830/83761. Email: humanethics@auckland.ac.nz.‖

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 30 July 2013 for (3) years, Reference Number 10032

Appendix B: Advertisement

Volunteers needed for a study on a computer program visualiser

PURPOSE OF STUDY:

This experiment evaluates the usability of a visualiser application in assisting novice learners of embedded microcontroller systems.

AS A PARTICIPANT, YOU WILL BE ASKED TO:

- test, edit and debug short programs using a visualiser application and talk about the process while you are doing it

- answer two questionnaires used to measure your experience in programming and using the application

The experiment will take approximately 90minutes

To be eligible you must be confident in English, have limited programming language experience, and should be aged 16 years or over. You do not need to have any experience with embedded systems. If you participate in this study you will be reimbursed with a $20 Countdown voucher for your time and effort.

If you wish to participate, or have any questions, please contact

Bill Collis wcol001@aucklanduni.ac.nz

Chair contact details: —For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Research Office, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 87830/83761. Email: humanethics@auckland.ac.nz.‖

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS

COMMITTEE ON 30 July 2013 for (3) years, Reference Number 10032

Appendix C: Consent form

THE UNIVERSITY OF AUCKLAND
FACULTY OF ENGINEERING
Department of Electrical and Computer Engineering

Science Centre (Building 303)
38 Princes Street
Auckland, New Zealand
Telephone 64 9 373 7599 ext. 88158
Facsimile 64 9 373 7461
www.auckland.ac.nz

The University of Auckland
Private Bag 92019
Auckland 1142

CONSENT FORM

THIS FORM WILL BE HELD FOR A PERIOD OF 6 YEARS.

Project title: Development of a visualiser application for novice students of embedded systems

Researcher: Bill Collis

I have read and understood the Participant Information Sheet. I understand the nature of the research and why I have been selected to participate in this research. I have had the opportunity to ask questions and have them answered.

- I agree to take part in this research
- I understand that I am free to withdraw participation at any time, and to withdraw any data traceable to me for up to two weeks after taking part in this research.
- I understand that all of the data collected from the survey will be non-identifying.
- I understand that I will be recorded (AUDIO ONLY) during part of the tasks.
- I understand that I will need to fill out a questionnaire at the beginning and the end of the task
- I understand that a third party who has signed a confidentiality agreement will transcribe the audio recordings
- I understand that I can choose to review the transcript and edit it if I wish
- I understand that only the researcher and his supervisors will have access to the questionnaire and observation data.
- I understand that the data may be used in publications about the tool.
- I understand that data will be kept for 6 years, after which they will be destroyed.
- I understand that Drs Gerard Rowe and Partha Roop have given assurance that my participation or non-participation will not affect my grades or relationships within the University, and that I can contact the head of department, Professor Zoran Salcic (Room 303.244, ext. 87802), if I wish to make a complaint about this assurance not being upheld.
- I do / do not wish to receive a summary of results.

my email address is _____

Appendix C

Name: _____

Signature: _____ Date: _____

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 30 July 2013 for (3) years, Reference Number 10032

Appendix D: Programming and Embedded Systems Experience Questionnaire

1. Programming and Embedded Systems Experience Questionnaire

Participant Code:	
Please describe any experience you have had with embedded (microcontroller) systems	
Please list any programming languages you have used and mark the circle you feel expresses your level of experience	
Language	Beginner ○ ○ ○ ○ ○ ○ ○ Expert
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
Please list any computer programming environments /IDE you have used and give your level of experience	
IDE	Beginner ○ ○ ○ ○ ○ ○ ○ Expert
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
	○ ○ ○ ○ ○ ○ ○
Have you used any program simulator or emulator before: YES NO If so, what was is called?	

Appendix E: Visualiser experience questionnaire

3. Visualizer experience questionnaire

Participant code:
To conclude please rate the following aspects of the visualizer. (please mark a circle below)
Visualizer contained the needed information to do the tasks (nothing was missing)
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:
The Visualizer contained just the right amount of information (not too complicated)
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:
The visualizer made the way the software and hardware worked together make sense
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:
At the end of the tasks the visualizer seemed natural and familiar to use
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:
The visualizer made programming more meaningful for me
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:
Visualizer words, images and diagrams were organized in a student friendly way
Agree Disagree
○----------○----------○----------○----------○----------○
Comment:

Appendix D

General comment on the visualizer
Please comment on the tasks: What was too easy? What was too challenging? Was it aimed too high, too low for you?
Task 1:
Task 2:
Task 3:
Task 4:

Appendix F: Semi-structured interview

The interview begins with a demonstration of the visualiser using the controlled pedestrian crossing block diagram and the automatic code generation features.

Questions were selected from the following depending upon the flow of the discussion.

- Tell me what aspects of embedded systems /programming you teach to novices?
- What are the things that novice students find difficult?
- What issue do they have with the hardware-software interrelationship?
- What software issues do they have?
- What are some of the surface and deep learning issues?
- What do students tend to forget after a course?
- How can visualisation help learners?
- How close is the visualiser to a real embedded system?
- What things are wrong with the visualiser?
- What things might a visualiser hinder in students' learning?
- What visualisations have you used? Own or others?
- How effective have they been? – Why?
- How would this visualiser fit with the schemes of learning in embedded systems here?
- What features would it need to have to be useful to you?
- What configuration files would be useful to help you?
- What tasks/exercises could be added to it?

Appendix G: data coding

Student data has been coded against four different aspects: the seven characteristics of good models (RQ1) in Table G.1, understanding of embedded systems (RQ2) in Table G.2, depth of thinking about programming (RQ2) in Table G.3, and whether a change to the visualiser, System Designer or task is required in Table G.4.

Characteristics in brackets e.g. (complete) indicates visualiser/task has a negative impact	
Complete	All needed elements and associations for learning a concept are present.
Concise	A summary only capped at five or so tasks to avoid excess information
Coherent	Sense making, showing interactions and rules for those interactions
Concrete	Depiction should be familiar to the learner
Conceptual	Shows the system operation meaningfully
Considerate	Student level vocabulary and organisation
Correct	Major analogies used are correct

Table G.1: Coding for RQ1, the characteristics of good models

Understanding in brackets e.g., ES-(HSI) indicates visualiser/task has a negative impact	
Hardware-software interaction	ES-HSI
Reactive to external environment	ES-R
Program state	ES-State
Concurrency with external constraints	ES-Concurreny
Criticality of the context	ES-Critical

Table G.2: Coding for RQ2, learners understandings of embedded systems

ST-textual	Writing of code – syntax level (ES-HSI)
ST-action	Program code carries out actions ES-HSI
DT-application	The code does something
DT-problem	Systematic programming applied to problem solving
DT-context	Insight that learning programming is empowering

Table G.3: Coding for RQ2, learners surface or deep thinking about programming

Changes to the visualiser indicated	VC
Changes to System Designer indicated	SDC
Changes to a task indicated	TC

Table G.4: Coding for changes indicated to the application or tasks

Responses of interest when coding the results

- student focus on text without appreciation of its action
- student focus on getting the program to work as opposed to a line of code to work
- time taken to carry out a task or one aspect of a task
- time between reading an instruction and correctly carrying out some action
- amount of movement through code searching for something
- incorrect or trial and error actions
- aimless changes to program code
- time reading instructions
- whether reading of instructions or carrying out actions came first
- amount of re-reading of instructions
- any 'aha' comments
- correctness of terminology used
- questions asked
- any assistance requests
- verbal comments made.

Student A:

Knightrider task

- 4:50 Given instructions
- Runs visualiser
- 5:20 Copied and pasted to end of sequence, runs first time – ST-textual
- Code modified until pattern is correct DT-application

Controlled pedestrian crossing task

- 8:45 started reading code
- 11:22 made code changes without running the visualiser ST-textual
- 14:40 Issue with changing speed of the running program VC
- 1520 moved 2 lines of code from beginning to end that he didn't need to – ST-Textual

- 16:30 issue switch clicking delay – VC- in step mode a push button switch is released between steps
- 17:30 checked with me that he had finished ST-action
- Code modified until pattern is correct DT-application

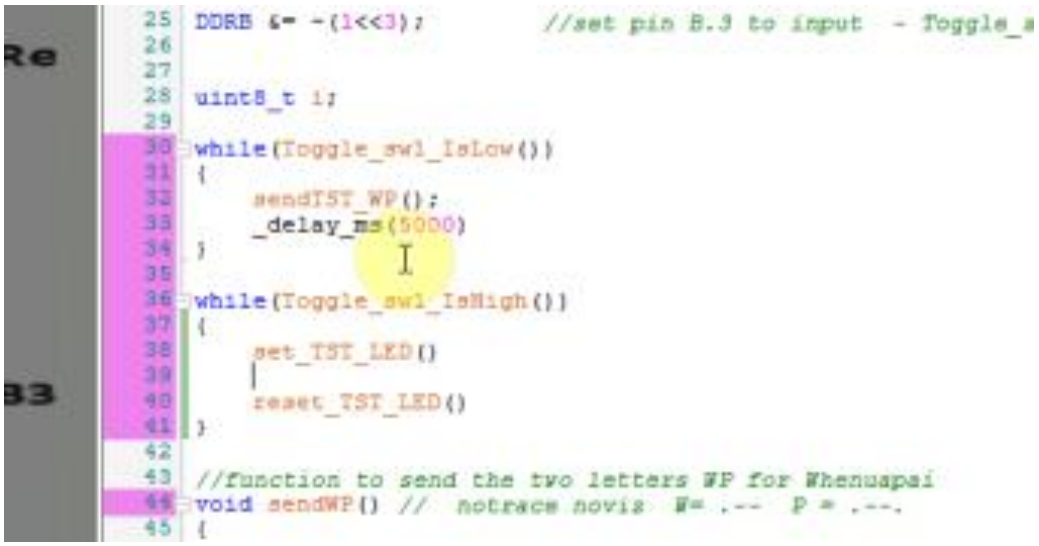
EMDR light bar task

- 17:50 read instructions, runs code in visualiser DT-application
- 20:35 can it step backwards?, instruction on break points ST-action
- 21:30 inserted correct line of code to turn off LED coherent ES-State
- 24:17 instruction given on breakpoints
- 24:57 copied and pasting large block of code DT-application
- Inserted break point himself to identify ST-action coherent
- 26:26 Trying to edit code without stopping execution VC – need to flag user visualisation has stopped
- Edited all pasted code lines correctly for PORTD coherent complete conceptual
- 29:20 question about end of sequence and number of LEDs DT-application ES-State
- 30:20 removed breakpoint, "where is the LED connected?" "oh I can see it there", complete conceptual ES-HSI
- Notices inconsistency in pattern with LEDs not working (due to error in DDR config) ST-action complete coherent conceptual ES-HSI
- Re reads instructions DT-application
- 31:40 finds error in DDRB and fixes it ST-action ES-HSI
- Comments on length of exercise TC – no need for exercise to cover all three ports, reduce to two
- 34:49 pointed his attention to the error window (considerate) (complete)
- 35:22- asks about how to write a delay function, asked about hash include for library DT-application ES-R
- Realises he has finished and code is correct DT-application

VOR Morse code task

- Opens visualiser without reading instructions ST-textual
- Read code ST-textual
- 38:20 changes zoom and run delay, familiar with controls very quickly considerate

- 39:00 asks question, guided to close visualiser to reread instructions ST-action
- Adds test LED to code
- 40:41 closes visualiser to reread instructions DT-application
- Starts to modify *sendWP* has partially understood the instruction; instructions to add new function and have switch between the two. ST-action
- 46:40 question about shift operator code in macro ST-action
- 45:40 question about how to combine code, created second while loop after the first change the first to while low, no enclosing while(1)ST-action



- 46:25 created second full function *tstwp* ST-action
- 46:44 closed visualiser to check Morse code for TST DT-application
- 50: realises mistake with code reintroduces while(1) and switches to if–else conceptual DT-application ES-R ES-State
- 51: rechecks instructions DT-application complete
- At end has not reused WP – TC- exercise needs prompting to make as much use of code reuse as possible ST-textual
- 53:30 "it's not turning off when the…" ES-HSI complete ST-action
- 54:40 combines the two while loops because he wanted "it to keep going" DT-application
- Realises he has finished and code is correct DT-application

Backyard alarm task

- starts to read instructions DT-application (prior to looking at code)
- 57:00 starts visualiser, begins to read code ST-action
- 57:45 where are the diagrams for … ST-action

- 58:30 goes back to block diagram to read DT-application TC- change layout of task to make easier to go backwards and forwards between visualiser and the task instructions
- 59:20 first run
- 59:50 tries reset button several clicks, tries enable switch ST-textual
- 1:00:50 can it auto-indent code? (considerate) VC- add auto indent
- Clicked auto-generate code button VC - move to submenu
- 1:02:50 Clicks reset switch several times looking for the code to read the switch concise complete ST-action ES-R ES-HSI
- 1:04:50 Finished code - used two if statements if (high){} and if (low) ST-textual
- Clear understanding of if(){} plan
- Realises he has finished and code is correct DT-application
- 1:07:20 "I think it helped because I didn't know how C handled working microchips, coherent concrete ES-HSI making it actually physically do things, all the stuff I've done has been handling written user input and output stuff understanding how that works and also the concept of having a, like I'd never thought of having a while loop as a 1, just to continually run it through and that was good" ES-R complete conceptual

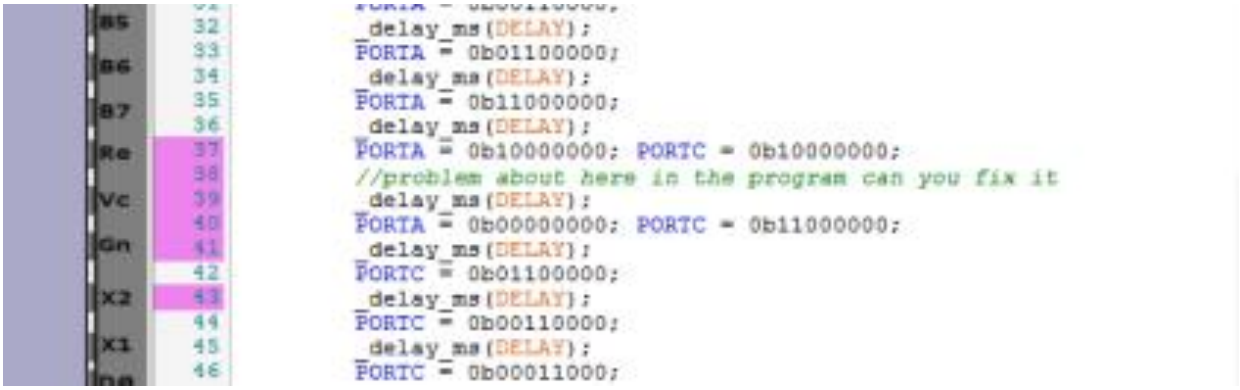
Student B

Knightrider task

- 4:00 added 3 LEDs to make 8 sequence concrete SDC - change to drag and drop interface
- 05:00 added into middle of sequence, not at end, copied and pasted, only changed comments at end of the line less than ST-textual
- 07:00 runs program "Oh" (realises mistake) and edits commands ST-action ES-HSI
- 08:00 realises some lines of code are only 7 bits edits these, ES-HSI complete coherent conceptual correct ST-action
- Code modified until pattern is correct DT-application

EMDR light bar task

- 09:35 runs program begins to edit code ST-action
- 17:00 VC - issue with code centring in window not LHS
- 18:55 changes DDRB correctly to DDRC ES-HSI ST-Action complete

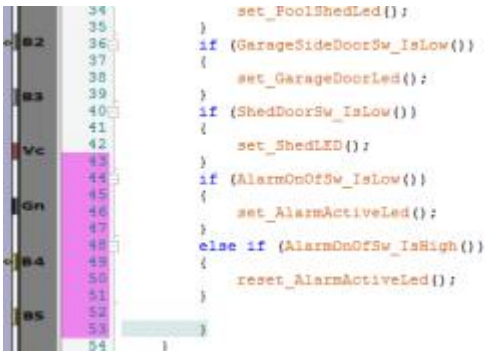


- Realises he has finished and code is correct DT-application

Backyard alarm task

- 30:00 reads instructions DT-application
- 31:00 opens visual reads instructions and code DT-application
- 36:00 add sequentially ST-textual
- 37:30 added else if inside {} inside if
- 40:00 still problems with wrong {} VC - needs auto indenting of code

- 41:05 adds missing { to if
- 45:00 adds master switch to code (not integrated with existing code) ST-textual



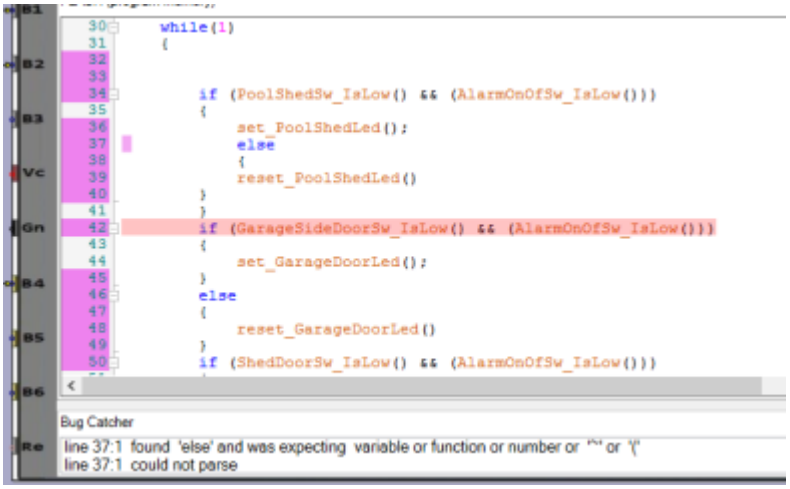
- 48:30 Identifies bug with else-if after single stepping program complete coherent conceptual
- 51:02 creates a duplicate test ST-Textual (ES-State)

```
//Program starts here
while(1)
{
    if (AlarmOnOfSw_IsHigh())
    {
        reset_PoolShedLed()
        reset_GarageDoorLed()
        reset_ShedLED()
    }
    if (PoolShedSw_IsLow())
    {
        set_PoolShedLed();
    }
    if (GarageSideDoorSw_IsLow())
    {
        set_GarageDoorLed();
    }
    if (ShedDoorSw_IsLow())
    {
        set_ShedLED();
    }
    if (AlarmOnOfSw_IsLow())
    {
        set_AlarmActiveLed();
    }
    else
    {
        reset_AlarmActiveLed();
    }
```

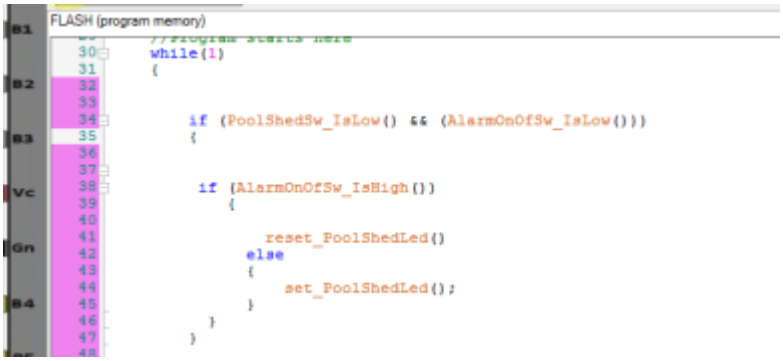
- Pointed towards flowcharts, “”that’s helpful” DT-application, begins to recode to merge the tests, still some duplication and has removed the resets for the LEDs ST-action (ES-State)

```
if (PoolShedSw_IsLow() && (AlarmOnOfSw_IsLow()))
{
    set_PoolShedLed();
}
if (GarageSideDoorSw_IsLow() && (AlarmOnOfSw_IsLow()))
{
    set_GarageDoorLed();
}
if (ShedDoorSw_IsLow() && (AlarmOnOfSw_IsLow()))
{
    set_ShedLED();
}
if (AlarmOnOfSw_IsLow())
{
    set_AlarmActiveLed();
}
else
{
    reset_AlarmActiveLed();
}
```

- 55:00 Adds resets but for second time forgets to close { before else



- Reads instructions – ignores error window DT-application
- 56:50 manipulates switches to see if it works but still in error conceptual coherent correct
- 57:10 I point out error and he fixes it VC- needs a stronger indication of error
- 1:02:00 trying to replicate code from flowchart ST-textual, struggling to turn application into code



- 1:03:00 forgets closing brace for if for 3rd time ST-textual
- "It's easy to see what happens because of the LEDs and switches, it's hard to get it to do what I want it to do" ES-HSI DT-application Complete Coherent Conceptual

- removes && changes merge and nest to just nest ST-action
- 1:06 for the 4th time removes { after if ST-textual

- 1:07:30 adds missing { then removes it again ST-textual

- Realises he has finished and code is correct DT-application

Student C

Knightrider task

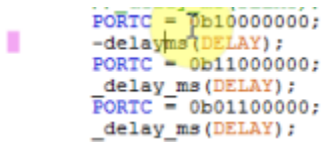
- 9:13 struggles to understand the nature of the sequence asks question about what it should do ST-textual (ES-State)

```
while (1) // repeat forever
{
    PORTA = 0b00000001;  //A.0
    _delay_ms (500);
    PORTA = 0b00000010;  //A.1
    _delay_ms (500);
    PORTA = 0b00000100;  //A.2
    _delay_ms (500);
    PORTA = 0b00001000;  //A.3
    _delay_ms (500);
    PORTA = 0b00010000;  //A.4
    _delay_ms (500);
    PORTA = 0b00001000;  //A.3
    _delay_ms (500);
    PORTA = 0b00000100;  //A.2
    _delay_ms (500);
    PORTA = 0b00000010;  //A.1
    _delay_ms (500);
    PORTA = 0b00010000;  //A.5
    _delay_ms (500);
    PORTA = 0b00100000;  //A.6
    _delay_ms (500);
    PORTA = 0b01000000;  //A.7
    _delay_ms (500);
```

- 10:00 cuts and pastes code into the program ST-textual complete coherent concrete
- 13:20 realises the bit patterns are incorrect ST-action complete coherent concrete ES-HIS
- Realises he has finished and code is correct DT-application

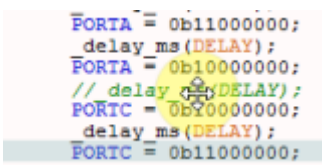
EMDR light bar task

- 16:00 instructed what to do. Less than ST-textual
- 20:38 extra bit in a the number ST-textual complete coherent concrete ES-HSI
- step through line by line
- 25:20 “can I use logic on the pins?” ST-textual (ES-State)
- Had to be directed to change the DDR error (complete) ST-textual
- 28:30 Consistently ignores errors even though highlighted bug catcher window in red, I suggest to fix them (complete)
- Struggles to see error in the line



```
PORTC = 0b10000000;
_delay_ms(DELAY);
PORTC = 0b11000000;
_delay_ms(DELAY);
PORTC = 0b01100000;
_delay_ms(DELAY);
```

- Expects the delay command to turn off the lights ST-textual



```
PORTA = 0b11000000;
_delay_ms(DELAY);
PORTA = 0b10000000;
// delay_ms(DELAY);
PORTC = 0b10000000;
_delay_ms(DELAY);
PORTC = 0b11000000;
```

- Hits Ctrl-Z (undo) while parser is running or stepping causes the undo action to try to close the visualiser (correct) SDC undo issue
- Has trouble with visualiser delay and delay program command interactions, (complete) (coherent) (conceptual) correct

```
PORTA = 0b01100000;
_delay_ms(DELAY);
PORTA = 0b11000000;
_delay_ms(DELAY);
PORTA = 0b10000000;
// _delay_ms(DELAY);
PORTC = 0b10000000;
_delay_ms(DELAY);
PORTA = 0b00000000;
PORTC = 0b11000000;
_delay_ms(DELAY);
PORTC = 0b01100000;
_delay_ms(DELAY);
PORTC = 0b0011000;
_delay_ms(DELAY);
PORTC = 0b00011000;
_delay_ms(DELAY);
PORTC = 0b00001100;
_delay_ms(DELAY);
PORTC = 0b00000110;
```

- See the duplicate line ST-action complete coherent conceptual ES-State
- 38:49 is using terms correctly not referring to lights anymore "PORTA7 and PORTC7 and C6" ES-HSI conceptual coherent correct
- 41:00 struggling to understand what to get the code to do to make the transitions between ports I suggest doing what another student did putting 2 commands on the same line **(coherent) (conceptual)** (ES-State)
- 43:40 "AAAAH" understanding – VC sequence issue with delays ES-State

```
PORTA = 0b00011000;
_delay_ms(DELAY);
PORTA = 0b00110000;
_delay_ms(DELAY);
PORTA = 0b01100000;
_delay_ms(DELAY);
PORTA = 0b11000000;
_delay_ms(DELAY);
PORTA = 0b10000000; PORTC = 0b10000000;
// _delay_ms(DELAY);
PORTA = 0b00000000; PORTC = 0b11000000;
_delay_ms(DELAY);
PORTC = 0b0110000;
_delay_ms(DELAY);
PORTC = 0b0011000;
_delay_ms(DELAY);
PORTC = 0b00001100;
_delay_ms(DELAY);
PORTC = 0b00000110;
_delay_ms(DELAY);
PORTC = 0b00000011;
```

- R45:00 realises that some numbers are only 7 bits complete coherent correct conceptual
- 46:20 he asks if he has finished, I remind him that the end LEDs stay on (ES-HSI) (ES-State) (DT-application)
- 51:30 when running the program there is still a very faint glitch due to the commented delay but he doesn't recognise it. (ES-State) (DT-application)

- I refer him to the errors, even though the error has line number he thinks it refers to his current line – needs training on reading error messages
- Fixes the error

Backyard alarm task

- 51:00 explained instructions , runs program
- 55:20 “Like the red for false” complete coherent concrete conceptual ST-action ES-HSI ES-State
- 57:40 “What do I do?” Have to explain instructions to him ST-action
- “If switch is high turn off all the switches” ES-HSI (he refers to inputs not outputs)
- Codes the switches correctly using the flowchart to guide his code
- 1:07:00 “it’s really useful because working with other programs you have to compile recompile and then run it and then check what’s wrong with it whereas with this when I’m stepping through it I can see like the output here as well whereas in there it’s like harder to see what you’ve mistaken” complete coherent conceptual considerate ES-HSI ST-action
- Understands he has finished and code is correct DT-application

Student D

Knightrider task

- No issues adds code to end of sequence and then edits ST-textual ES-HSI conceptual coherent correct
- Correctly identifies and changes DDR error ES-HSI conceptual coherent correct
- 15:20 “that’s cool” coherent conceptual
- Realises he has finished and code is correct DT-application

Controlled pedestrian crossing task

- Runs visualiser
- 23:35 recognises what is wrong with the sequence
- 27:40 adds the commands to make the pedestrian lights work ST-action ES-HSI ES-State conceptual coherent correct
- 32:00 Refers to problem allowing time for the pedestrians to cross, changes comments to reflect problem and context DT-problem ES-Critical ES-State

- Checks he has finished and code is correct DT-application

Backyard alarm task

- 37:30 bypasses instructions "what is it supposed to do?" ST-textual
- 41:00 Adds two if statements ST-action ES-HSI (DT-application)

```
if AlarmOnOfSw_IsLow();
{
    while(1)
    {
        if (PoolShedSw_IsLow())
        {
            set_PoolShedLed();
        }
        if (GarageSideDoorSw_IsLow())
        {
            set_GarageDoorLed();
        }
        if (ShedDoorSw_IsLow())
        {
            set_ShedLED();
        }
    }
}
if AlarmOnOfSw_IsHigh();
{
    while(1)
    {
        if (PoolShedSw_IsHigh())
        {
            set_PoolShedLed();
        }
        if (GarageSideDoorSw_IsHigh())
        {
            set_GarageDoorLed();
        }
        if (ShedDoorSw_IsLow())
        {
            set_ShedLED();
        }
    }
}
```

- "What does while(1) mean?" ST-action to ST-application ES-R
- ignores error – I suggest to read the error

```
//Program starts here

if AlarmOnOfSw_IsLow();
{
    while
    {
        if (PoolShedSw_IsLow())
        {
            set_PoolShedLed();
        }
        if (GarageSideDoorSw_IsLow(
        {
            set_GarageDoorLed();
        }
        if (ShedDoorSw_IsLow())
        {
            set_ShedLED();
        }
    }
}

if AlarmOnOfSw_IsHigh();
```

Bug Catcher

line 31:1 expected: tOpenBracket found: tBitwiseNot
line 31:1 expected: tCloseBracket found: tSemicolon

VC-drop the ‘t’ in the error – still need a stronger indication of error

- No () around the test and inserts a ; at the end of the line cancelling the test
- Realises he has finished and code is correct DT-application

VOR Morse code task

- 55:42 VOR – cannot interpret written instructions, responds to verbal instructions ST-action
- Copies and pastes existing function ST-textual
- 1:07 no {} around if and else blocks, fixes this ST-action complete coherent concrete conceptual
- Adds test LED to only the test mode, corrects error ES-HSI ES-State complete coherent concrete conceptual ST-action
- “I learnt how to use functions, declare prototype before use” ST-action
- “so simple to see everything happening” ES-HSI
- Realises he has finished and code is correct DT-application

Lecturer A comments

- There are two aspects to learning embedded systems: peripherals and interfacing, the other is doing the design in C, I have tried to decouple the two to make the overall process easier for learners.
- Students want to do more of the design themselves rather than being constrained to given designs.
- The visualiser is simple to use, something that students can latch into immediately.
- Very good to have simple and full memory views.
- Names with registers not just abbreviation would be useful.
- Number systems and conversions between, implications of assignment between types, how to store real numbers on a computer, hex is difficult for students to understand.
- Demystify some of the nuances of embedded systems.
- Certainly useful for stage1, 2 & 3 students and has a benefit to a broad range of different classes of learners such as skilled programmer new to embedded systems.
- Tool is useful with different users, expert programmers who don't know embedded systems, naïve programmers who don't know embedded systems.
- Why include BASIC?
- Benefits: revealing registers, scope, casting, string typing, bitwise operations.
- Extend with Timers to microcontroller model.
- Pointers to structs is a difficult concept for students.
- Extend its benefits to include peripheral communication.
- Use a pipe and output to a console to get I/O outside the microcontroller model?
- Discussion relating to trade-offs between in-lining a macro and implementing a function.

Lecturer B comments

- Would be useful as a tool for a short introduction to all CS students.
- GUI would be useful to handle environmental issues: distributed communication, multiprocessor.
- C is much too hard for school students.
- Record simulations on the website as learning tools so students don't even need the software.
- Also useful for selling software as it has commercial value.
- How to teach the safety critical nature of embedded systems.

- Replacement of human skill by model based engineering is an important understanding captured here.
- Need exercises that make students think, school students are not trained to think about the deep difficulties involved in engineering.
- Uses outside education for companies training staff.
- Very good for all engineering students to learn using this.
- Interrupt issues are difficult for students to understand.
- Add other embedded systems constraints like power consumption e.g. sleep features.
- Extensibility/reconfiguration is very important issue, use different microcontroller models.
- Add assembler as well and show micro-architecture aspects.

Lecturer C comments

- Wow amazing use of context to help learning.
- Has a commercial future, I would use this with my students.
- Is C appropriate for school students to learn?
- Automatically generated code templates great way to help student over syntax issues.
- The views are great being closed up to start with so that the learner isn't confronted with the typically busy simulator environment.
- The voltage divider on the switch is very nice, helpful for students to see the voltage levels change and link the switch to input code.
- Powerful views of memory, simple view very good for novices.
- What about interrupt processing? That's a difficult one for students.
- What aspects of C cant it do? [structs] well they're not that different to arrays so it shouldn't be hard for you to code them.
- Character LCD is nicely slowed down to model a real character LCD.
- completing the graphics LCD emulation would be useful.
- Add simulator for serial IO / 1-wire.

Lecturer D comments

- Very positive giving structure of code to students.
- The context of the exercises is different to most teaching that we do which is outside context.
- PICAXE product has shortcomings for tertiary use, this is a commercial system.

- The colouring of the syntax for Booleans was good, as it would help students identify the code sequence around if-then-end if.
- Use of state charts is superior to flowcharts when teaching embedded systems.
- Should sell it, it could be useful in many school classrooms and tertiary too.
- The different views of memory were brilliant, really liked the full view to explain floating point numbers.
- Scope and recursion are very difficult concepts, these were well demonstrated in the memory view and I would like to trial it with students.
- Write the code for a switch, that's a difficult one for new students.
- Finish the interrupts.
- Do a basic timer model like the other visualiser already done.
- Maybe model all registers, and add console I/O.
- Looking forward to seeing the class diagram and sequence diagram auto generate C++ code.

Appendix H: The existing System Designer application

I have developed and tested System Designer over the last four years with students. It contains a number of drawing types to support students' technological practice in the development of their embedded systems projects. In this appendix I introduce some of the drawing tools it incorporates.

The range of drawing tools and existing visualisations can be found in the code map (Figure H.1). System Designer has been written in C# and prior to this study incorporated over 30,000 lines of program code (excluding libraries and Visual Studio auto generated code).

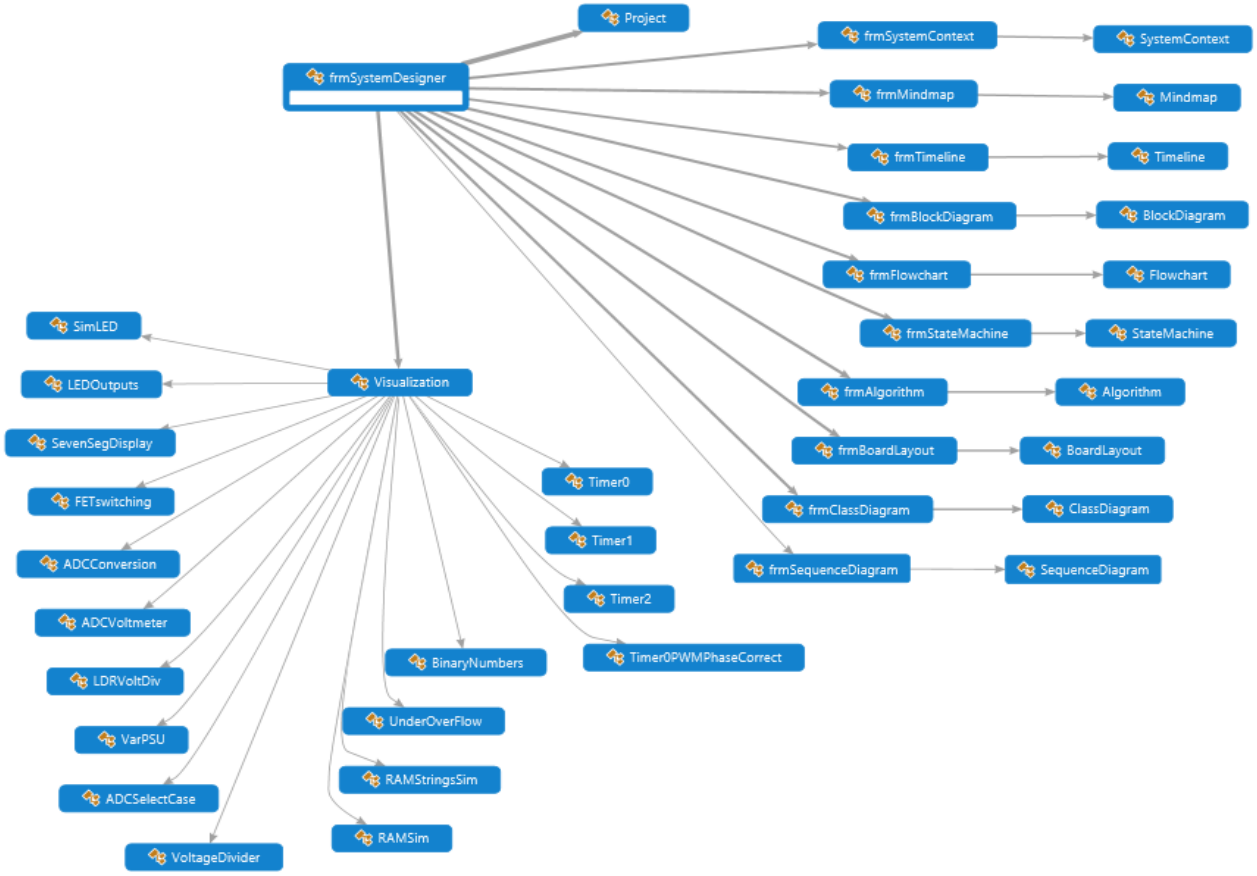


Figure H.1: System Designer code map for diagrams and visualisations prior to this study

System Designer makes extensive use of a diagramming library from Mindfusion.eu to handle the drawing of nodes and links. System Designer was initially envisioned as a tool to support students integrating the wide ranging complexities of their project work. Specific goals

investigated in action research within the classroom include: its ability to provide a model driven methodology to support development of system software, to automate/template code generation so as to focus students away from syntax difficulties.

Systems thinking

It is essential even with small embedded systems projects that a methodical process is followed (Koopman et al., 2005); this process oriented approach is the underpinning concept of the socio-cultural / participatory technological practice that students undertake in New Zealand secondary school technology education. The key driver behind the development of System Designer was for an integrated platform for student work that created a single environment that allowed students to see the different parts of their project as joined rather than disparate.

System context diagram

Brief Development Achievement Standards in NCEA require students to undertake analysis of their client and stakeholder's environments. A tool to develop system context diagrams based upon aspects of requirements engineering was developed to support this aspect of student work (Figure H.2).

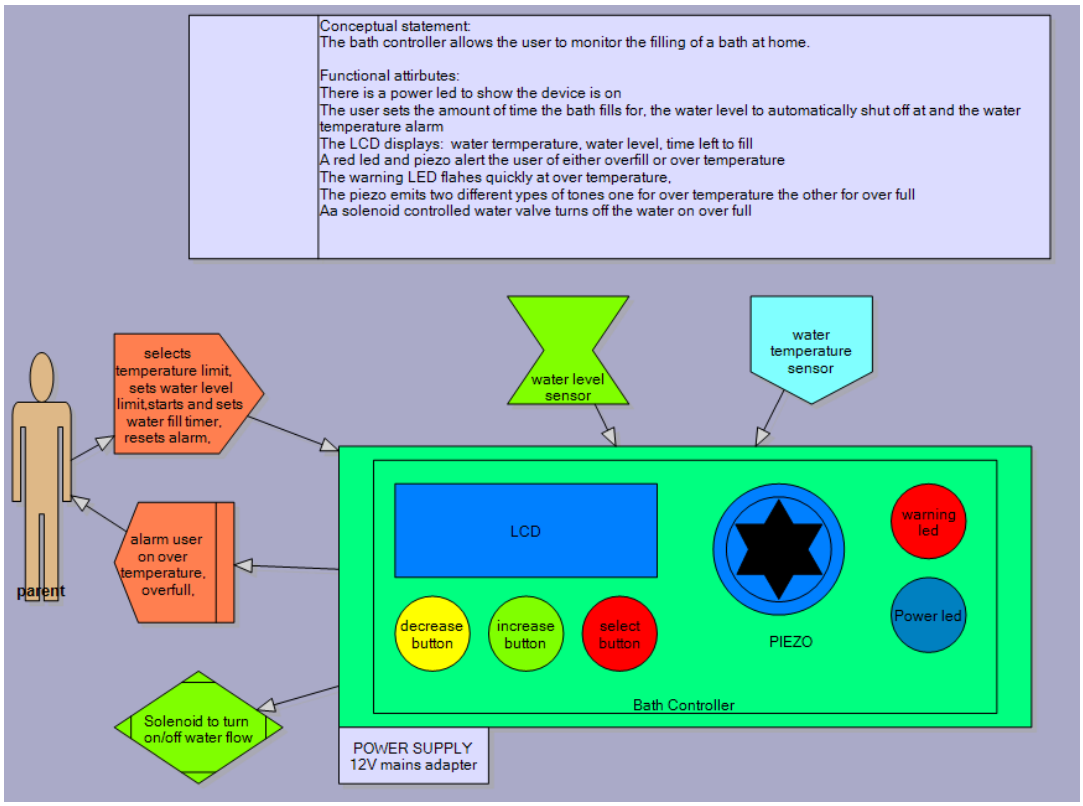


Figure H.2: System context diagram

Model driven design

It has become increasingly important in industry to rely on model-based design (Koopman et al., 2005). The tools that do this though are highly complex and sometimes expensive. The following models were developed with the goal of being sufficient for novice learners in terms of cognitive load theory and to provide transitionary support into more complex modelling that they might come across later in tertiary education.

Block diagram

Whilst students use a CAD program for designing PCBs (EAGLE from CADSOFT), a hardware block diagram is the main conceptual modelling tool used in design of electronic systems and standard throughout industry. A Hardware Block Diagram tool (Figure H.3) is the central diagram relating to student development work. It has a drag and drop interface to allow students to easily create block diagrams for their designs and tabulates all the connections automatically. Initially a microcontroller is selected for the student's project (various AVR microcontroller types are made available via an XML file). Various input and output devices can be connected to it. As these are connected the connections are stored in tables on the diagram. The tables are initially hidden to reduce cognitive load but are made visible via a toolbar button.

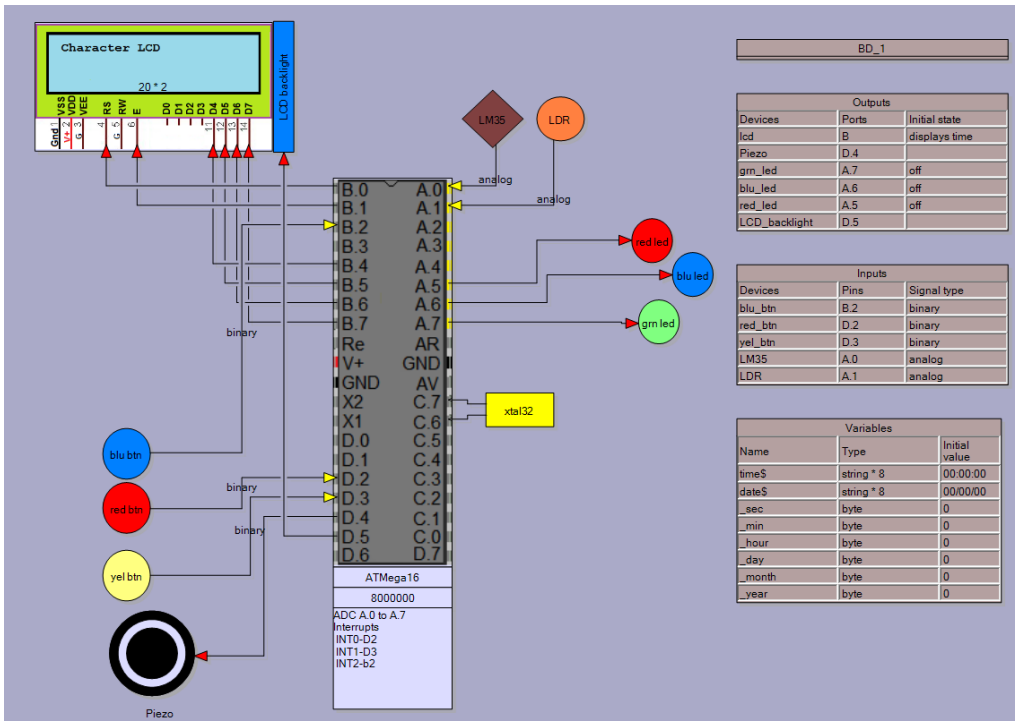


Figure H.3: Block diagram for an alarm clock created in System Designer

Flowcharts

The flowchart designer (Figure H.4) existed prior to this study however it has been redeveloped as a live coding design. On the left of diagram is the flowchart. As programming constructs (while, if, do, etc.) are dragged from the toolbar and dropped onto it, the program code window on the right of the diagram is automatically updated. The table in the middle of the figure captures the I/O devices from the block diagram editor. A context dropdown menu for each flowchart element allows selection of code hints based upon I/O and variable items in the table. Output code hints relate to output devices and variables, and Boolean tests relate to variables and input devices.

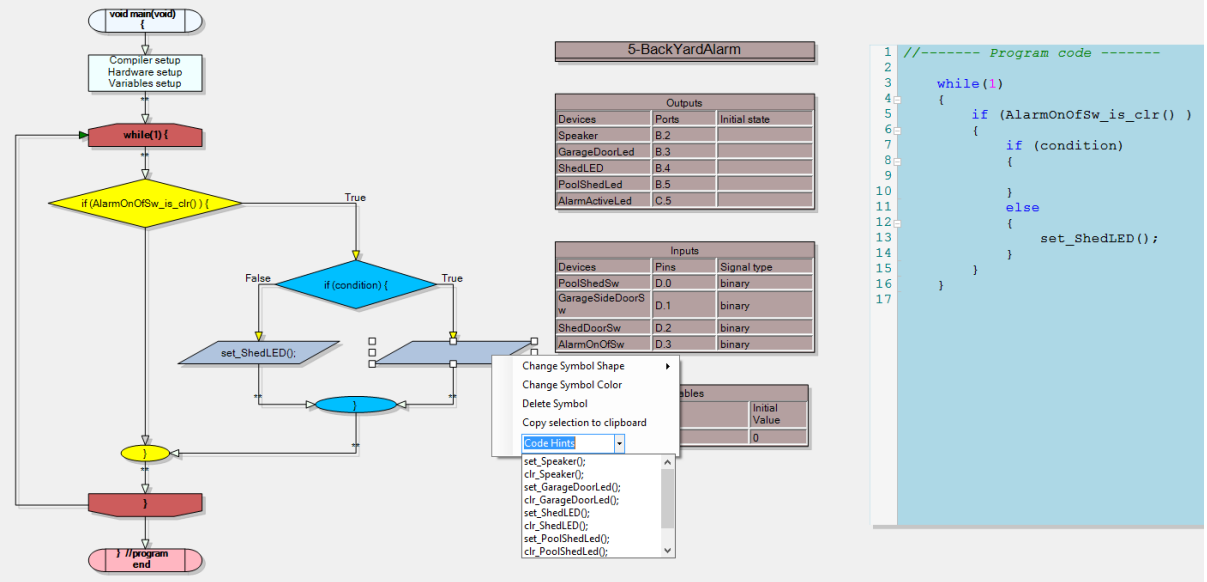


Figure H.4: Flowchart diagram editor

State machines

The state machine editor (see Figure H.5) allows students to model state machines and automatically generate program code for them. An additional modelling option is the ability to capture reactive elements of embedded systems via timers and pin interrupt devices in the state machine.

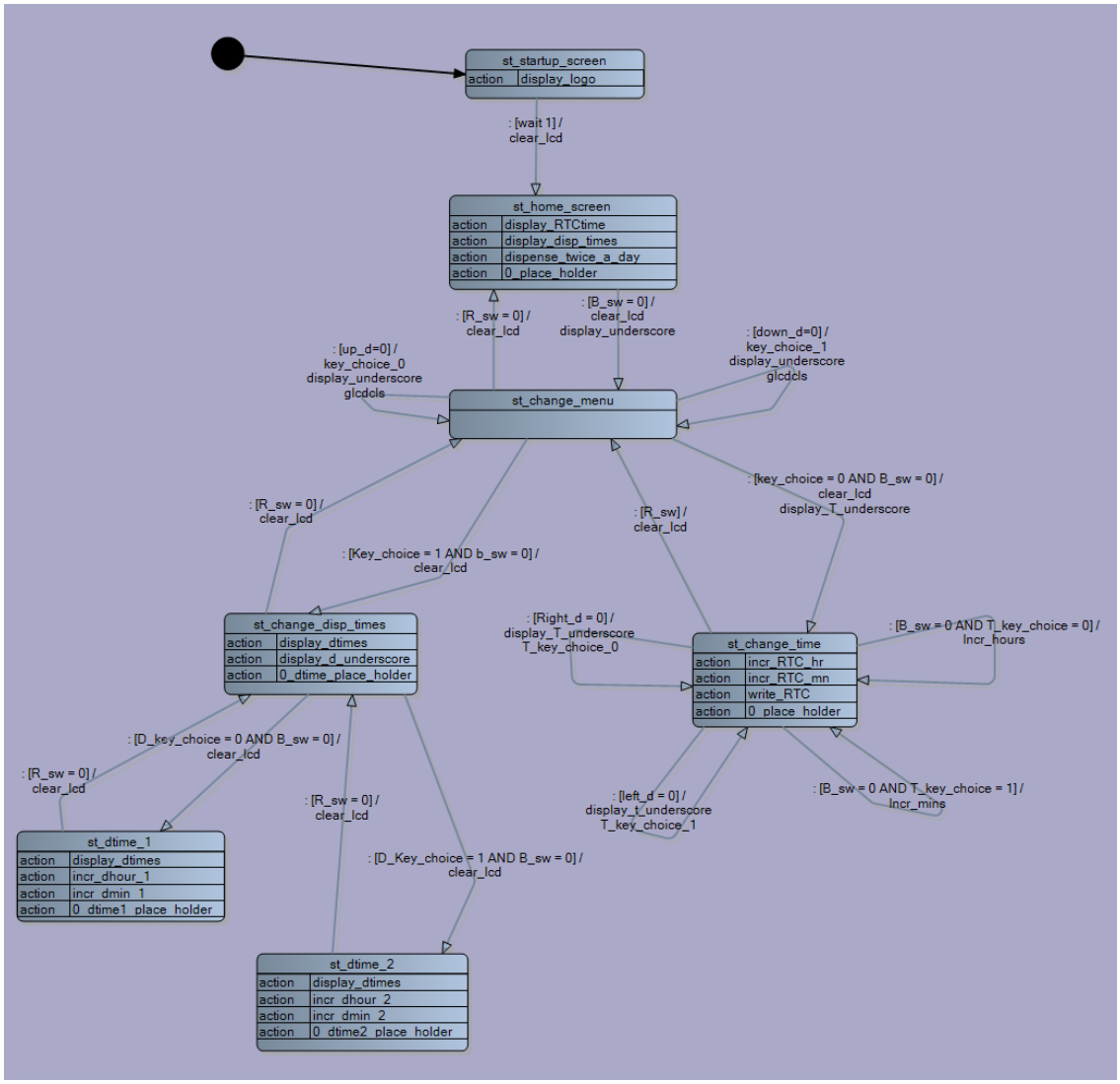


Figure H.5: State machine for a student developed process controller

Syntax management

One of the largest and most frustrating challenges for novice programmers is correct syntax, as identified in many significant studies (Abrantes et al., 2008; Adcock et al., 2007; Brusilovsky & Sosnovsky, 2005; Denny et al., 2011; Gomes & Mendes, 2007; Kelleher & Pausch, 2005; McCracken et al., 2001), because it hinders them moving on to the more important aspects of schema development and planning in programming. Automating code generation is a feature of many of the diagrams in System Designer allowing students to reduce the time spent concentrating on syntax and allowing them cognitive space to focus on design.

Documentation strategies

Whilst novices generally have poor documentation strategies (Letovsky & Soloway, 1986) students enjoy using a computer to draw diagrams and fill in templates. There are planning and documentation requirements in NCEA that students need to fulfil for their projects; to facilitate this System Designer has mindmapping software and timeline creation software. A generic mind map template, Figure H.6, can be automatically generated. It contains a typical sequence students follow when undertaking a full project; with the numbers in () in each part of the diagram representing the time to be allocated in weeks for that aspect of the project.

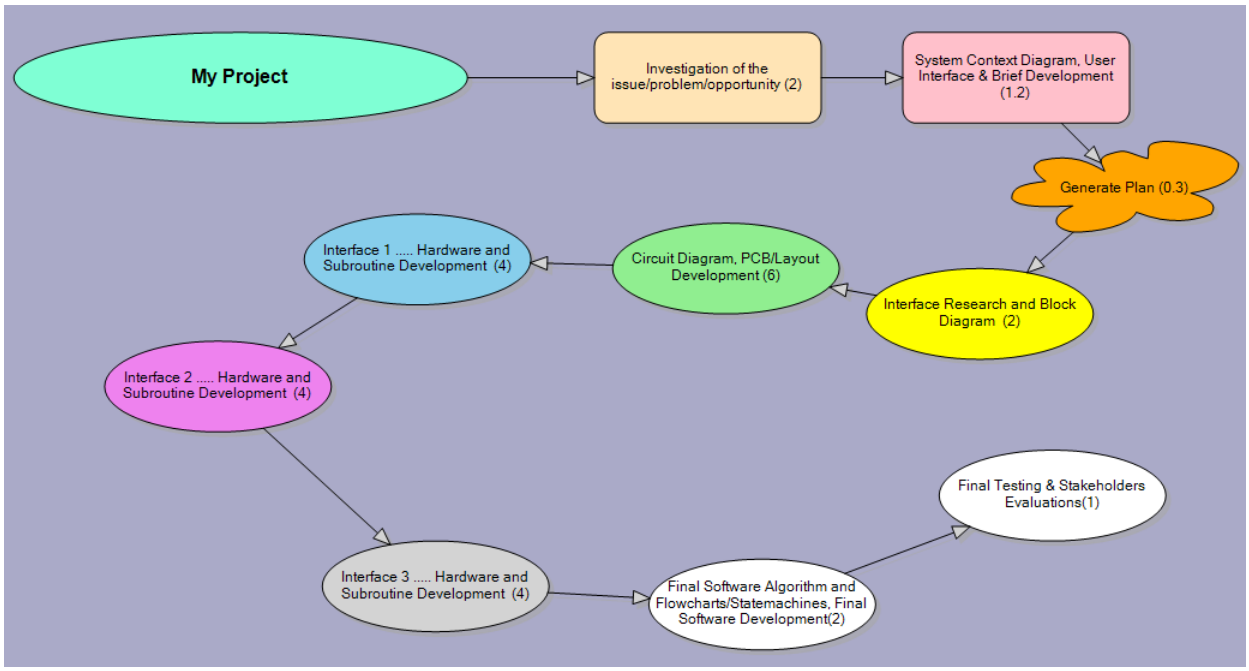


Figure H.6: Mindmap overview for a student's project

After the mindmap is modified to suit each student's project it can be automatically turned into a timeline, into which specific detail for each stage can be logged (Figure H.7).

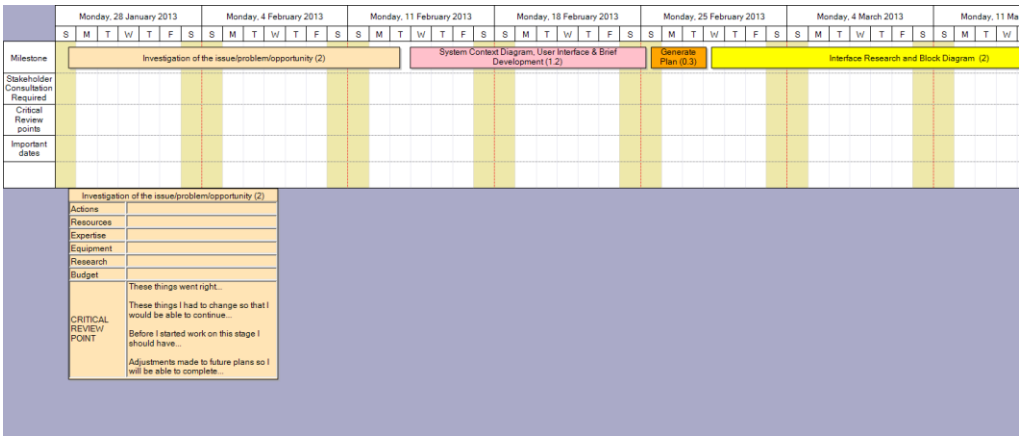


Figure H.7: Automatically generated timeline

Class Diagram tool (alpha)

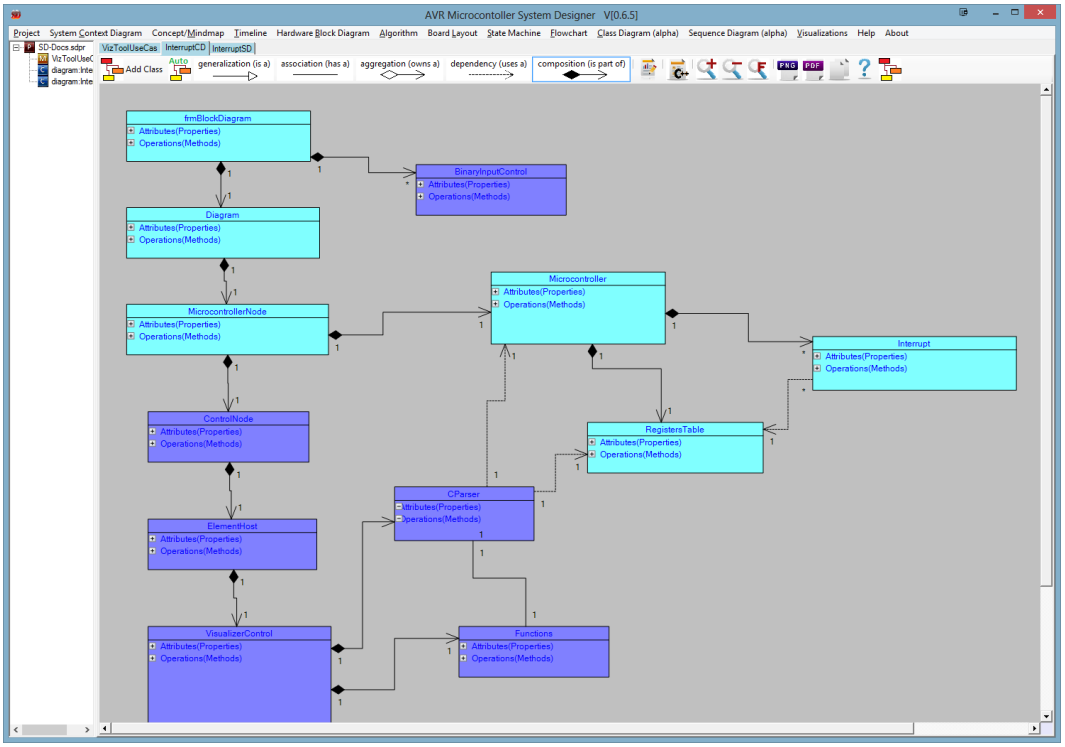


Figure H.8: Class Diagram Editor

This tool (Figure H.8) automatically generates classes and C++ code from block diagrams as part of user interface class development. The next stages are to have it generate boundary classes from

a use-case diagram (developed by the user in the system context diagram tool) and an internal processing class (developed by the user in the state machine tool).

Sequence Diagram Tool

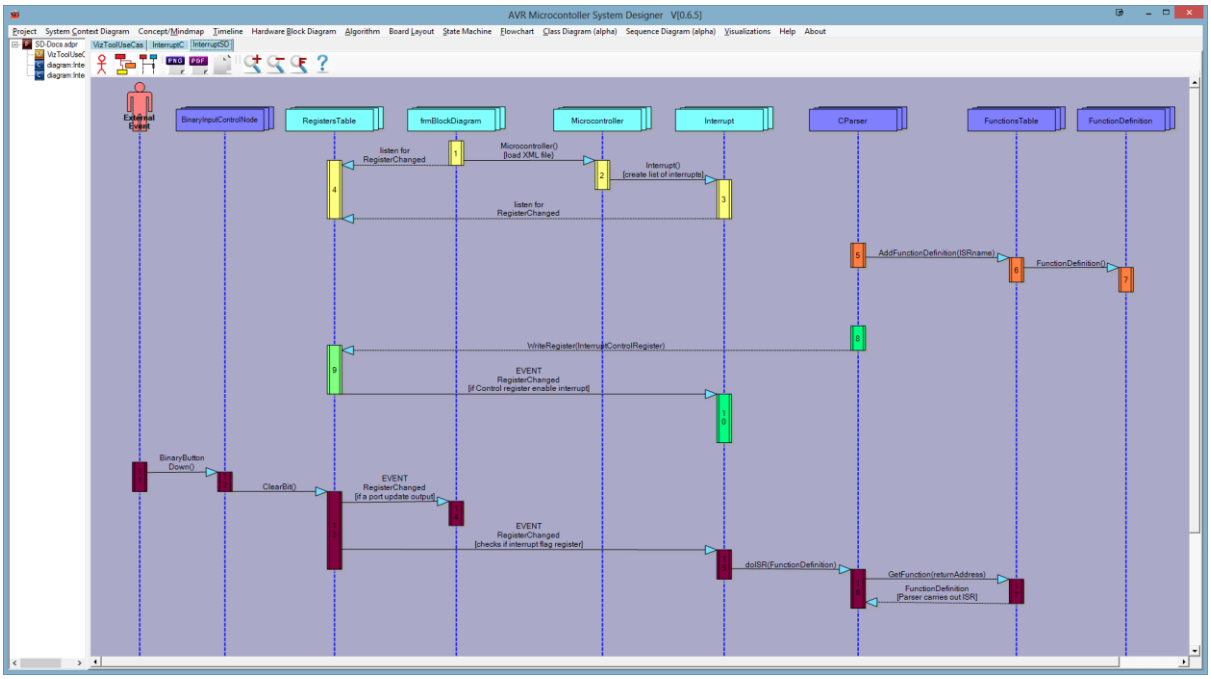


Figure H.9: Sequence Diagram Editor showing early planning for visualiser interrupt processing

This tool (Figure H.9) allows the user to design interactions between objects. It will at some stage become integrated as part of the automatic code generation.

Appendix I: Technical details of the Visualiser extension

The visualiser is a set of classes that have been used to extend the original System Designer application; these exist only when the visualiser is open. The visualiser is attached to the block diagram tool as presented in the simplified code map in Figure I.1. The frmBlockDiagram class contains a MindFusion Diagram object which holds the MicrocontrollerNode and associated IO devices (switches, LEDs, 7Segment displays, analog inputs, keypad, character and graphics based LCDs). The frmBlockDiagram class was extensively modified as part of the development of the visualiser.

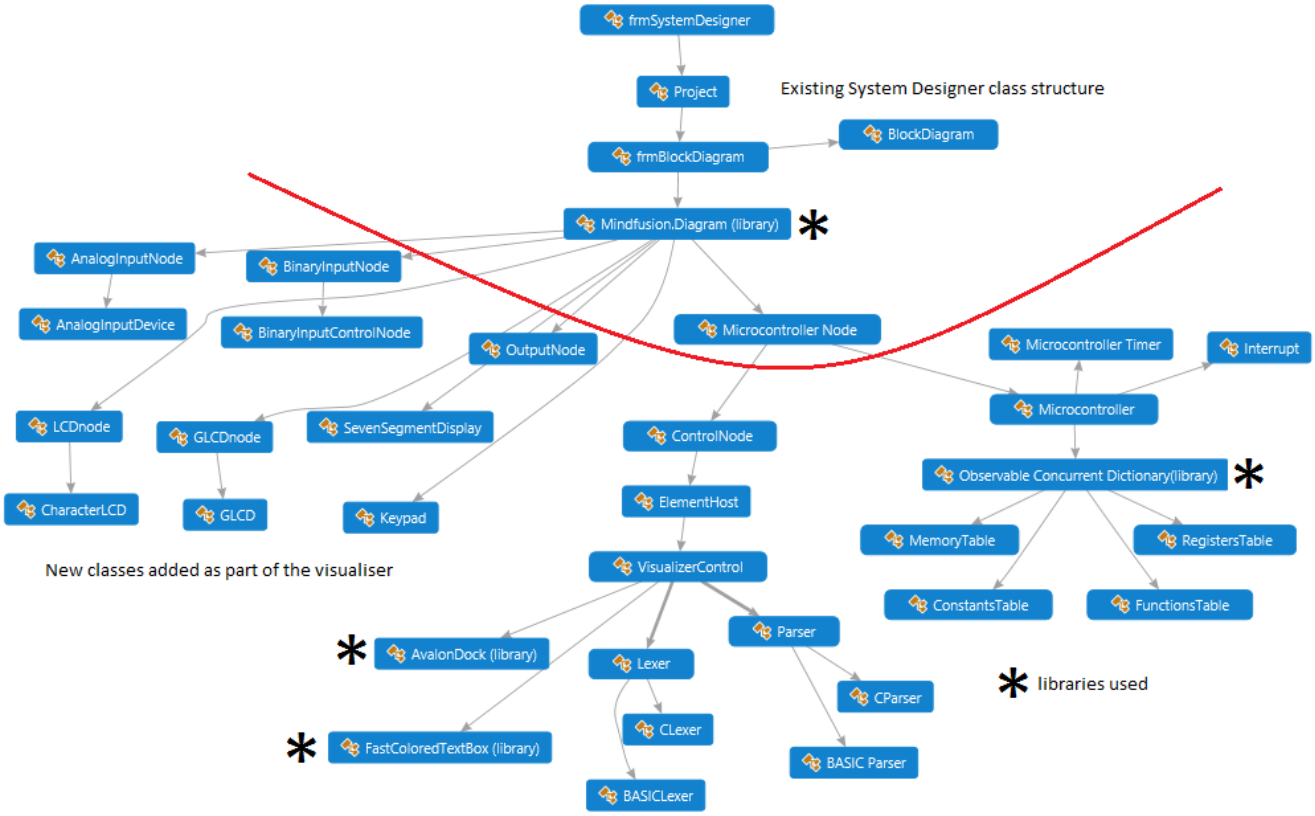


Figure I.1: System Designer codemap

Before beginning the work on the visualiser, the block diagram tool already contained a simple microcontroller class that held details about the microcontroller being used on the diagram. This section describes the further development of this class that was required to implement details about the visualisation process, and one of the most complex aspects of the visualiser, the processing of microcontroller interrupts.

Figure I.2 is a class diagram (drawn using the ClassDiagram tool within System Designer) of the interclass relationships involved in visualising interrupt processing. The two colours of the classes represent those classes that are part of a standard block diagram (in light blue) and those that exist only while the visualiser is open and running (the darker blue colours).

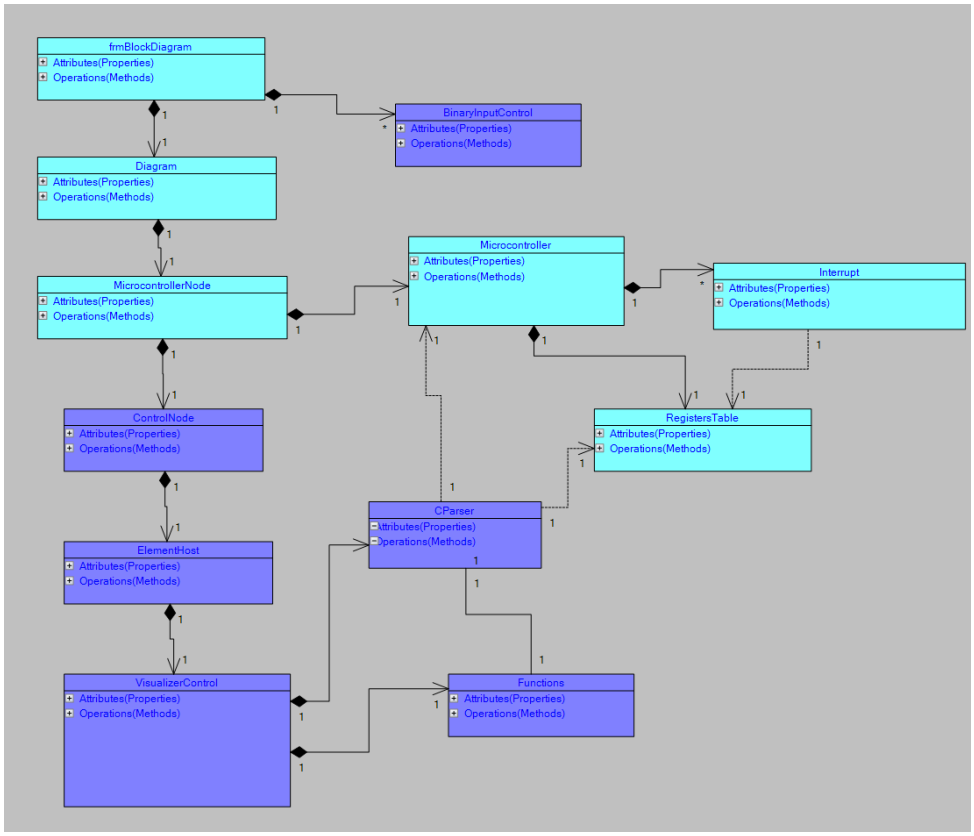


Figure I.2: Classes within the block diagram relating to interrupt processing

Interrupt processing

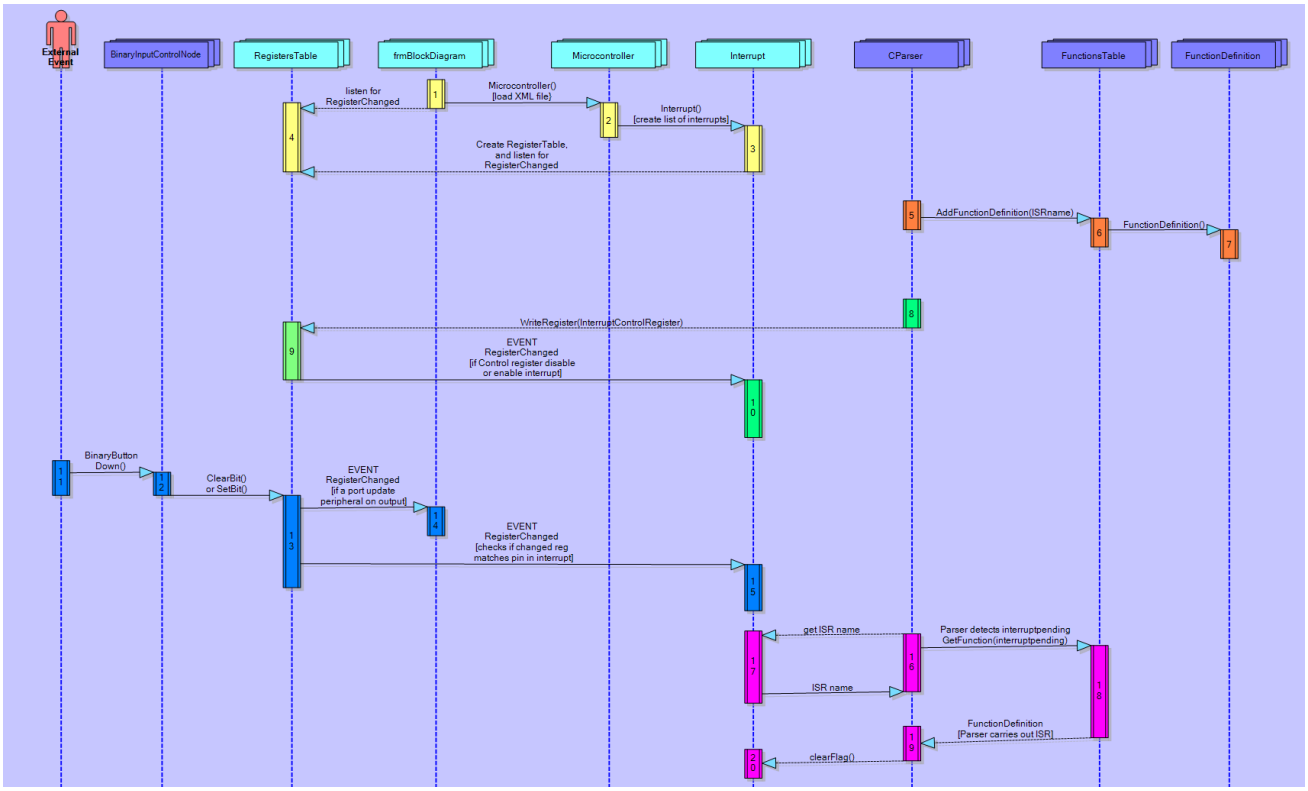


Figure I.3: Sequence diagram for interrupt processing

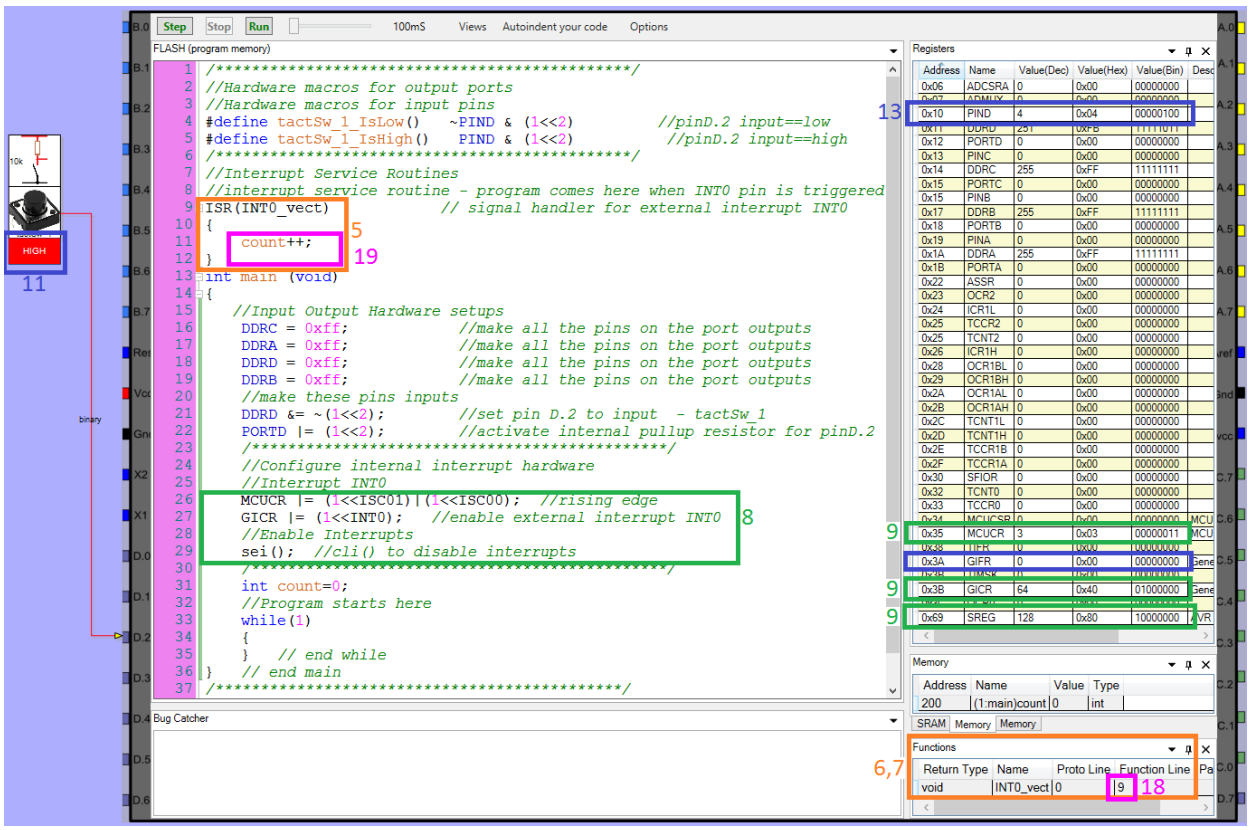


Figure I.4: User interrupt based program

Figure I.3 and Figure I.4 show the four stages of visualising the processing of interrupts (Figure I.3 was drawn using the sequence diagram tool in System Designer).

Stage1 (Figure I.3 yellow): When a microcontroller node is created within a block diagram (#1), two XML files are opened and details about the specific part number for the microcontroller and associated interrupts are loaded. This includes creating an instance of a Microcontroller class (#2) which has a List object that holds a number of Interrupt objects (#3). At this time each instance of Interrupt registers to 'listen' for changes to values in the RegistersTable (#4).

Stage 2 (orange): During parsing of a user program, any interrupt service routine encountered will be added to the functions table (#5,#6,#7).

Stage 3 (green): During parsing of the user program (Figure I.3 #8), the microcontroller registers (Figure I.4 #9) are configured to support interrupt detection (GICR) and to set the trigger type (MCUCR), as well as setting the global interrupt flag (SREG bit 7). Any change to a register will signal each instance of class Interrupt to compare the changed register to its control register (Figure I.3 #10). If the particular interrupt is enabled (GICR bit is set) it will then compare its control register setting (MCUCR bits 0 and 1 for INT0) with the change in state of the input pin register (PIND bit 2) to see if the incoming action equates to the setting. This might be a positive edge, a negative edge, a change or a high level. If there is a match then a Boolean flag called 'Triggered' within the Interrupt class is set.

Stage 4 (blue and pink): During parsing (Figure I.3 #12) if an input device connected to an interrupt pin changes (such as Figure I.4 #11, PIND.2/INT0 is activated by the tact switch), a change will be made to the PIN register for PIND (Figure I.3 #13). This fires the event RegisterChanged to the parser class (Figure I.3 #15). When the parser reads the next token in its list of tokens it will check to see if the global interrupt flag in SREG is set. If so it will interrogate each instance of Interrupt to see if its 'Triggered' flag has been set (Figure I.3 #16). If it finds a flag set it will get the ISR field from the interrupt function that matches the Interrupt (Figure I.3 #17). It will find the line number of the ISR function from the functions table (both figures, #18) and then jump to the ISR (both figures, #19). Then the parser will reset the Triggered flag in the interrupt (Figure I.3 #20).

Appendix J: Lexer detail

To make the implementation of the visualiser less complex two separate tasks are undertaken and a class created for each. The first task is the tokenising of the code in the Lexer class. The lexer turns program code into tokens (defined in a struct) which are then stored in a List.

A number of characteristics are featured in the lexer:

- Separation of keywords for C and BASIC
- Separation of constants into different types (integer, float, string).
- Error checking of number formation e.g. that a float conforms to applicable rules.
- White space is removed.
- Identifiers are classified as functions, constants, defines or variables. As these are identified they are stored into unique classes each built on a dictionary type for later reference in the lexer and the parser.
- Function prototypes, function calls and function definitions have different tokens attached to them so the parser can separate the parsing requirements of each.
- CR and LF tokens are separated and kept as breaks for the parser
- Comments are checked for structural correctness and then dropped
- Commands to the visualiser have been developed as discussed earlier
- Semicolons are captured (though currently ignored in the parser)

There is a base Token class, the BasicToken and CToken classes inherit from this.

Appendix K: C parser detail

The second stage in the visualisation process is the parsing of the tokens. The parser is a recursive descent parser (Figure K.1). The article *Let's build a Compiler!* (Crenshaw, 1988) was of invaluable assistance when beginning to write my own parser.

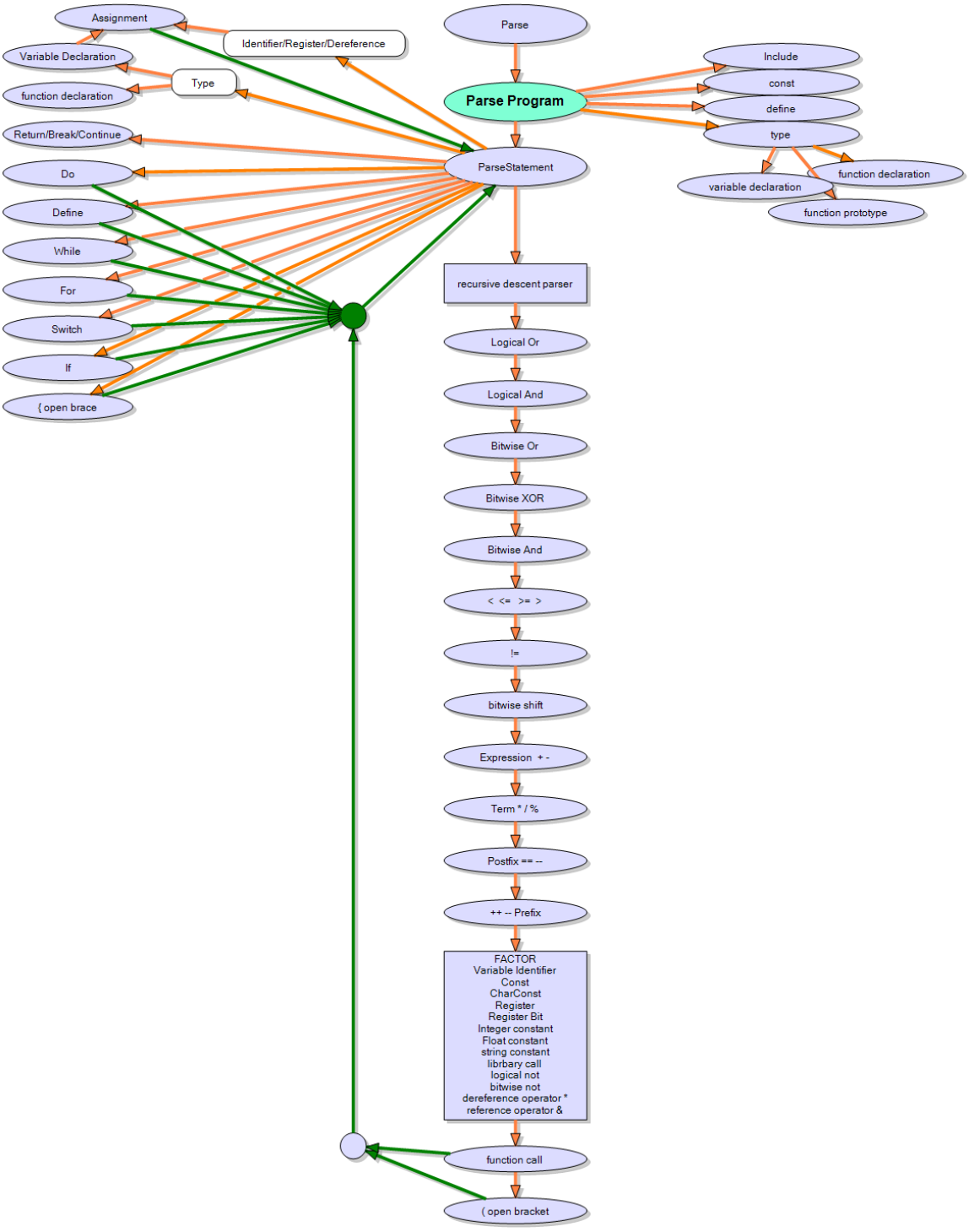


Figure K.1: C code recursive descent parser (recursive return paths shown in green)

Figure K.2 shows a tree representation from the parser for the line of code:

If (a<=b && b<=c)

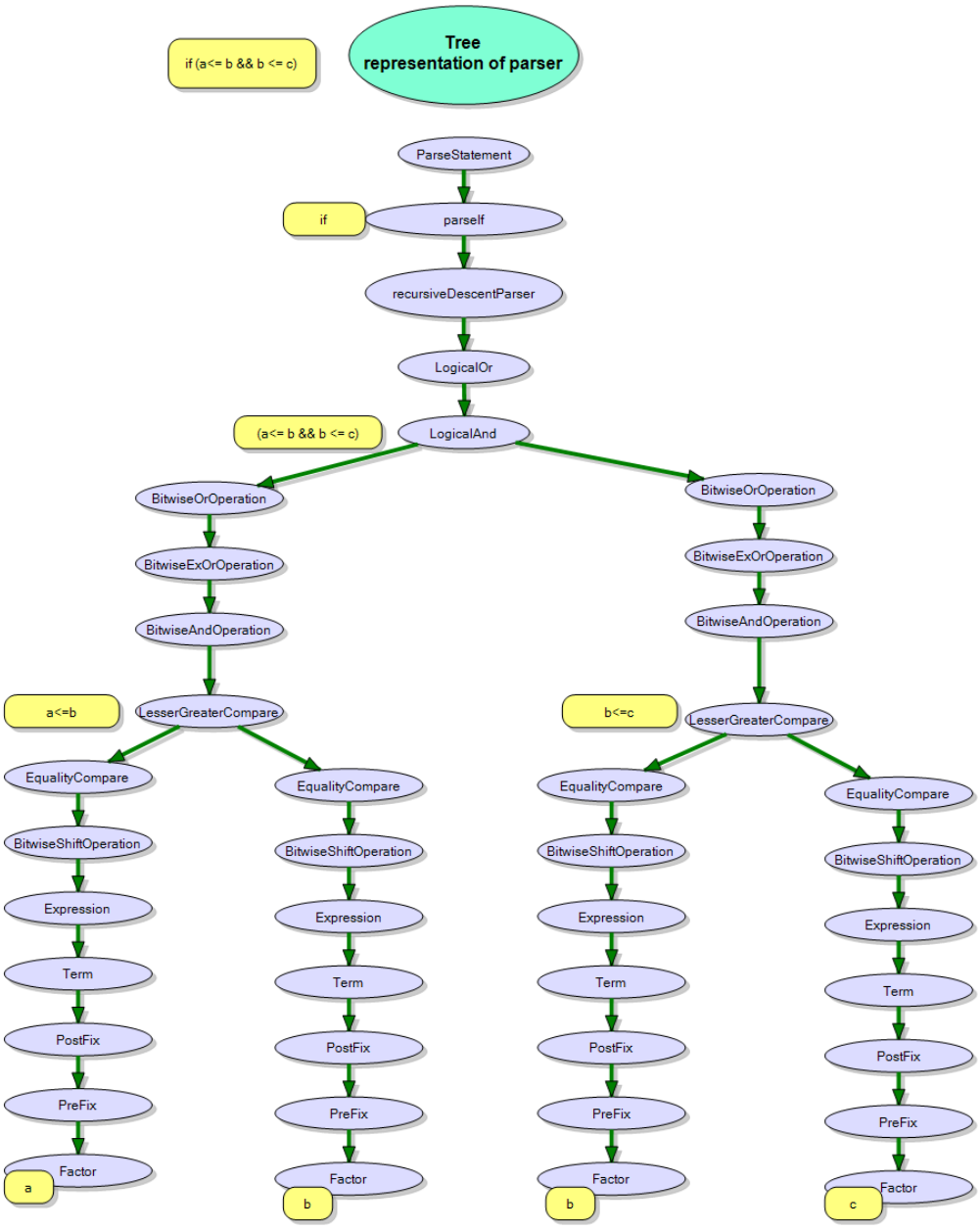


Figure K.2: Tree representation for parsing if(a<=b && b<=c)

Each method in the parser is called with 2 parameters a value and a type which are passed by reference e.g. **bool LogicalOr (out dynamic value, out VarType vartype)**

The usage of ‘out’ is pass by reference however somewhat different to ‘ref’. ‘Out’ informs the compiler that the value must be initialised inside the function while ‘ref’ means it must be initialised before the function call; while ‘ref’ is both ways, ‘out’ is out only.

The use of a dynamic value allows C# to pass any value without type checking at compile time.

Variable types

The different variable types available are captured in a struct and this is used to manage the different variable types of both BASIC and C. In C# integer variables are held as 32 bit signed integers so type checking after each stage of calculation in the parser is necessary to identify over/under flow operations on smaller integer types and indicate to the user that casting may be required.

The float is a single precision (32 bit) number using IEEE 754, single bit sign, 8 bit exponent and 23 bit mantissa.

```
public enum VarType
{
    none,
    uint8_t,
    int8_t,   //s8
    uint16_t,
    int16_t,
    uint32_t,
    int32_t,
    avrFloat,
    avrBit,
    avrString,
    ptr1,
    ptr2,
    ptr4
}
```

Parser polymorphic behaviour

The parser is split into a base class (Parser) and two derived classes CParser and BasicParser. Common methods and properties are in the base class.

Parser has one virtual method 'Parse()', which is overridden in CParser and BasicParser. When the visualiser instantiates an instance of parser, it calls the appropriate constructor for the type e.g. CParser() or BasicParser(); further calls to Parse() will go to the overridden method polymorphically.

CParser

C code for an AVR microcontroller will generally conform to a standard template, e.g.

```
includes
defines
consts
functions or function prototypes
global variable declarations
void main(void)
{
        hardware initialisation
        local variable declarations
        while (1)
        {
                Main program block
        }
}
```

Whilst the visualiser will recognise this template it is not mandatory, as when teaching novices it is important to reduce the amount of obfuscating information, so less complex statements can be parsed also e.g.

```
hardware initialisation
while (1)
{
        program block
}
```

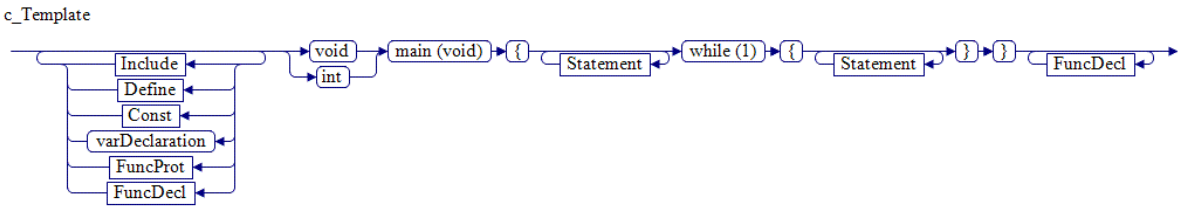
Appendix L: EBNF for the C parser

The recursive descent parser is implemented using a set of rules, which are expressed using EBNF notation and visualised using the software EBNF-Visualiser (“EBNF Visualiser,” 2013)

Typical C template rule for an AVR program expressed in EBNF:

```
c_Template =  {       Include
              |       Define
              |       Const
              |       varDeclaration
              |       FuncProt
              |       FuncDecl
              }
              ("void" | "int") "main (void)"
              {"
                     {Statement}
                      "while (1)"
                      "{"
                              {Statement}
                      "}"
              "}"
              {FuncDecl} .
```

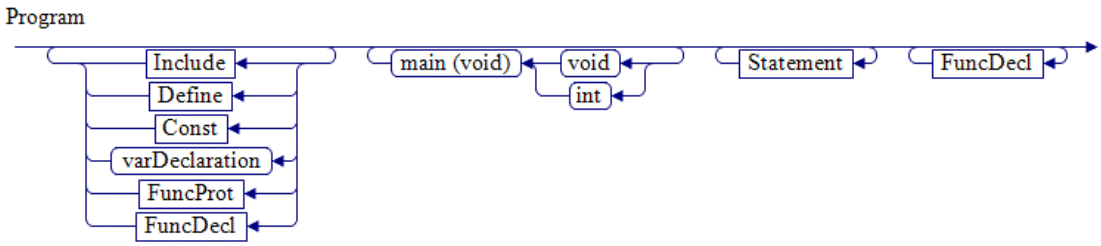
This is shown in diagram form via EBNF Visualiser as:



The program's actual implementation is slightly changed. This allows users to enter statements without the need for the complete syntax of a C main program being included and the obfuscating *int main (void) {}* expression being seen by novices.

```
Program =       {       Include
                |       Define
                |        Const
                |       varDeclaration
                |       FuncProt
                |       FuncDecl
                  } {  ("void" | "int") "main (void)"
                }
                {Statement}
                {FuncDecl} .
```

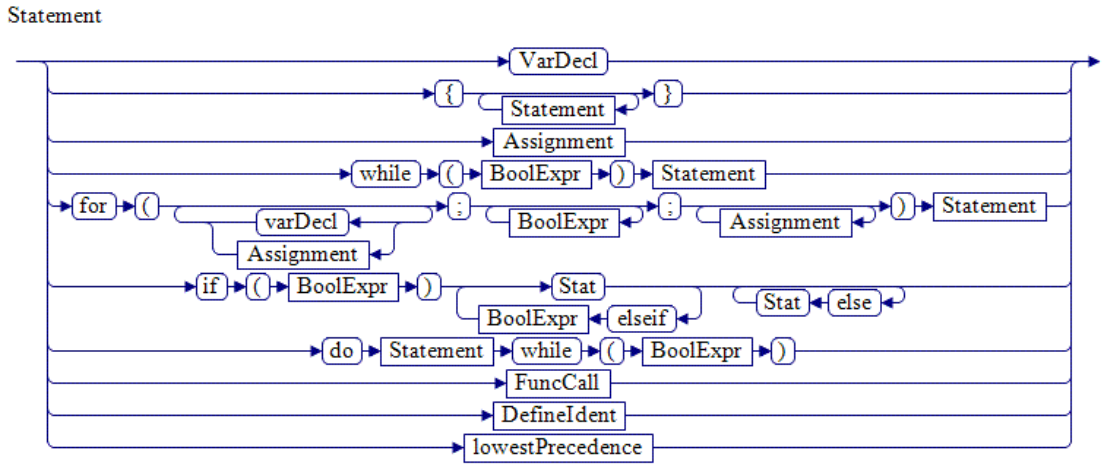
It is visualised as:



The state machine for ‘statement’ is:

```
Statement =     VarDecl \n
                |       "{" {Statement} "}"
                |       Assignment
                |       "while" "(" BoolExpr ")" Statement
                |       "for" "(" {varDecl | Assignment} ";" {BoolExpr} ";" {Assignment} ")" Statement
                |       "if" "(" BoolExpr ")" Stat {"elseif" BoolExpr Stat} {"else" Statement}
                |       "do" Statement "while" "(" BoolExpr ")"
                |       FuncCall
                |       DefineIdent
                |       ConstIdent
                |       recursivedescentparser .
```

RecursiveDescentParser reflects the start of the parser to interpret the program code.



In order of precedence (lowest to highest) the commands are:

```
recursivedescentparser =        BitwiseOrOperation.
BitwiseOrOperation =            BitwiseExOrOperation {'|' BitwiseExOrOperation }.
BitwiseExOrOperation =          BitwiseAndOperation { '^' BitwiseAndOperation} .
BitwiseAndOperation     =       BitwiseShiftOperation {'&' BitwiseShiftOperation }.
BitwiseShiftOperation =         Expression {( '<<' | '>>' ) Expression}.
Expression =                    ["+" | "-"] term {["+"|"-"] term } .
Term =                          Factor | {['*' | '/' | '%']  Factor} .
Factor =                        VarIdent
                                |       DefineIdent
                                |       ConstIdent
                                |       RegAddress
                                |       FuncCall
                                |       Number
                                |       '~'
                                |       '(' recursivedescentparser ')'.
```

The lexer separates into categories different identifiers; this was implemented so as to display to novices the many different aspects captured in the system but without having to work with a single complex symbol table. These are separated into: variables, registers, constants, defines and functions.

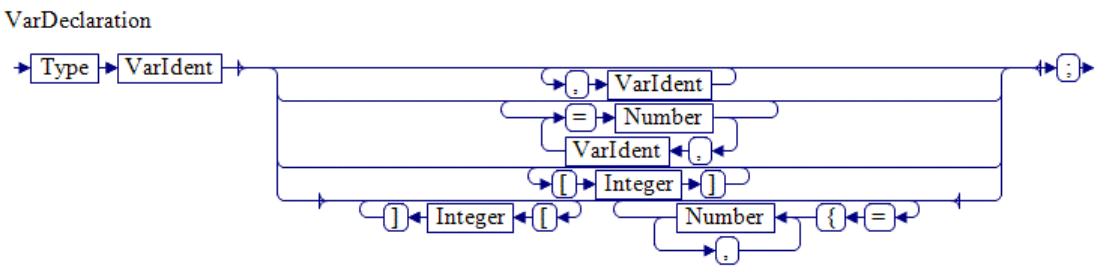
```
VarIdent =          {letter| "_" | "$" } {letter | digit | "_"| "$"}.
ConstIdent =        {letter| "_" | "$" } {letter | digit | "_"| "$"}.
DefineIdent =       {letter| "_" | "$" } {letter | digit | "_"| "$"}.
FuncIdent =         {letter| "_" | "$" } {letter | digit | "_"| "$"}.
RegAddress =        "PIN" letter
                    |   "PORT" letter
                    |   "DDR" letter.
```

A range of types are represented within the CParser

```
Type =          "void"
                | "char"
                | "uint8_t" |  "int8_t"
                | "uint16_t" | "int16_t"
                | "uint32_t" | "int32_t"
                |  "double"
                | "string" .
```

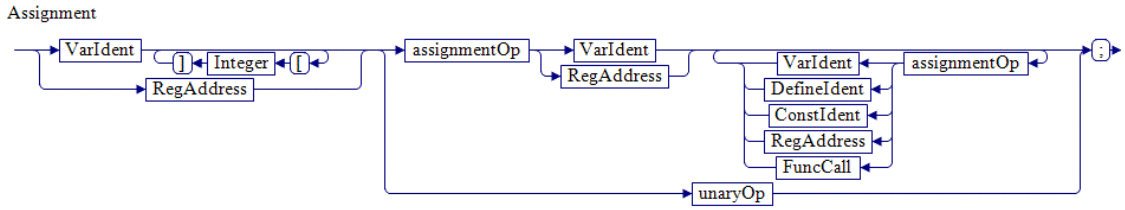
Variable declaration includes arrays

```
VarDeclaration =    Type VarIdent { "," VarIdent}
                    |       {"=" Number {"," VarIdent "=" Number}}
                    |       {"[" Integer "]"}
                    |       {{"[" Integer "]"} {"=" "{" Number {"," Number} }}  }";".
```



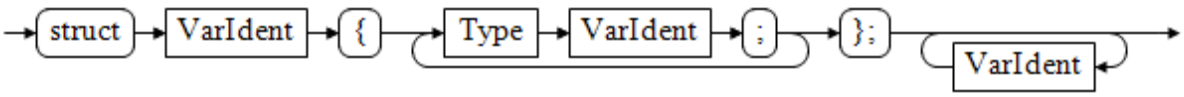
Assignment to variables

```
Assignment =        (VarIdent  {"[" Integer "]"} | RegAddress)
                        ((assignmentOp   ( VarIdent | RegAddress )
                        { assignmentOp   (VarIdent
                                | DefineIdent
                                | ConstIdent
                                | RegAddress
                                | FuncCall )  })
                    |       unaryOp ) ";".
```



Struct:

```
Struct = "struct" VarIdent "{"  Type VarIdent ";" {Type VarIdent ";"} "};" {VarIdent}.
```



Although there are several ways of declaring structs in C, the following pattern has been implemented (where the declarations today and tomorrow are optional)

```
struct weather {
  int windspeed;
  string winddirection;
  int temperature;
  int humidity;
} today, tomorrow ;
```

Appendix M: Visualiser run/step control

Run State

To start the visualiser the user either clicks the V button on the toolbar or right clicks on the microcontroller in the Block Diagram to open a context menu and select the visualiser (Figure M.1).

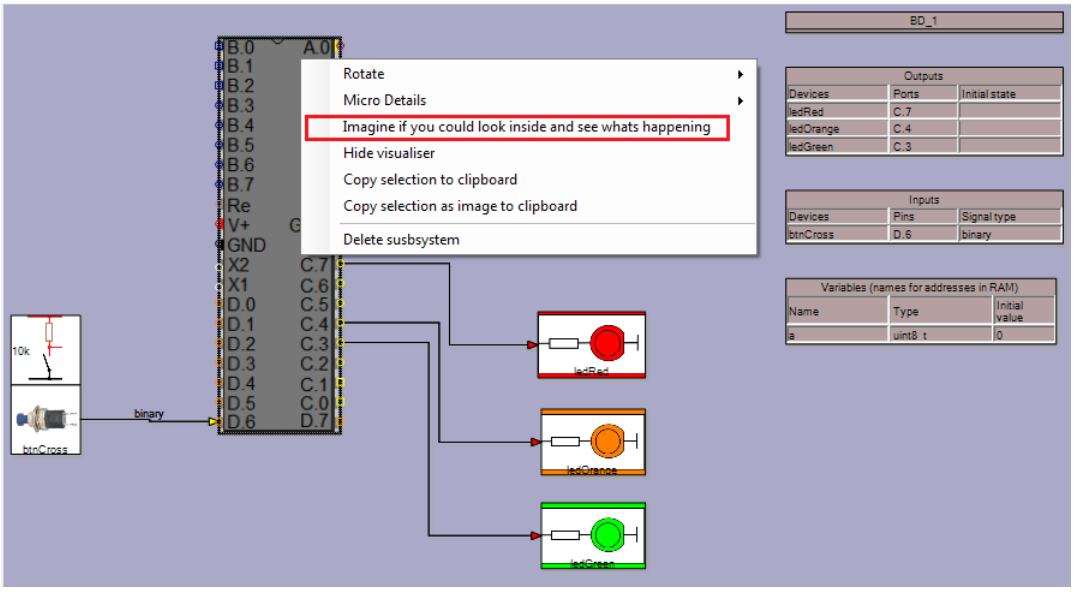


Figure M.1: Starting the visualiser in System Designer

When the user starts the visualiser the events in Figure M.2 take place: a Mindfusion ControlNode is created and attached to the Microcontroller node. Within the ControlNode an ElementHost is used as VisualiserCtrl is a Windows WPF UserControl.

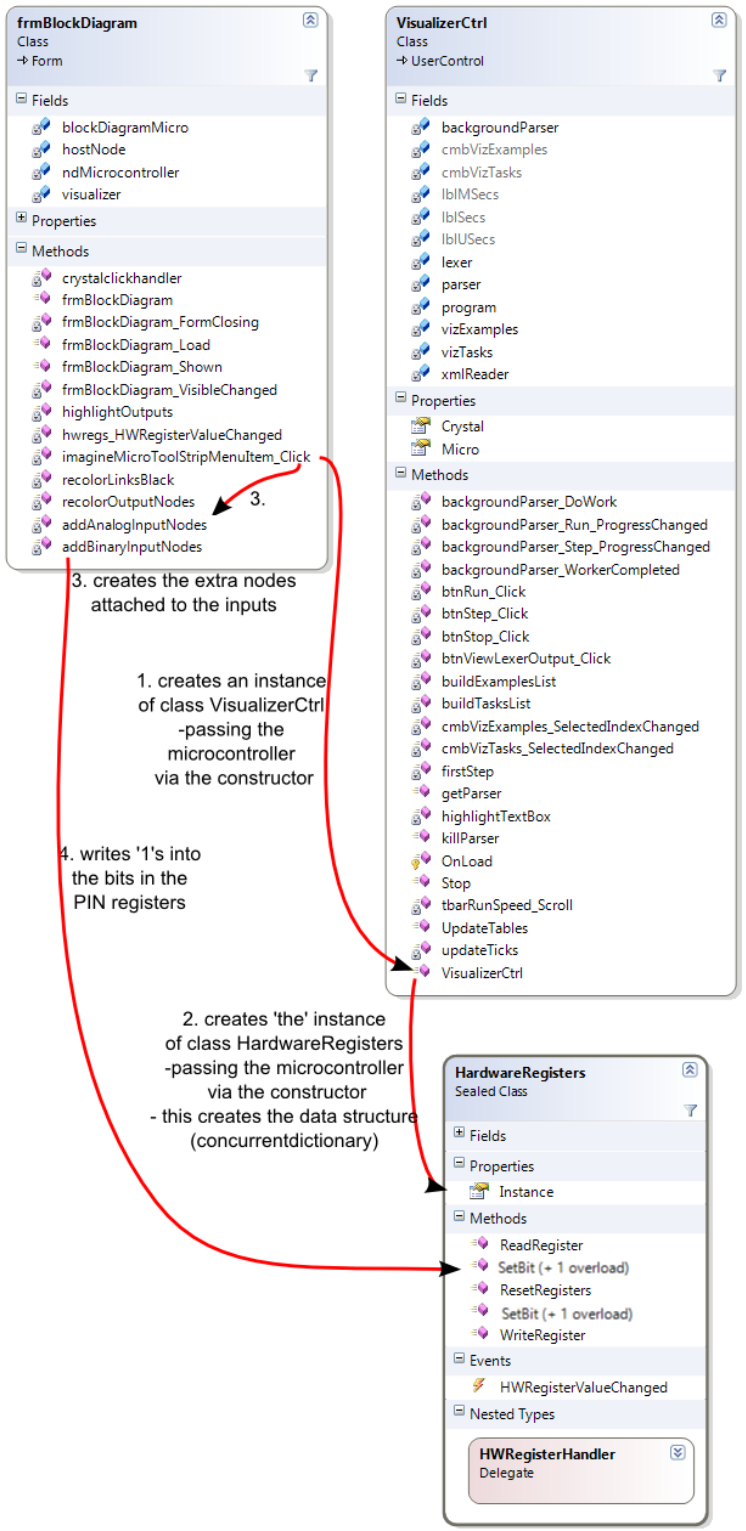


Figure M.2: Starting the visualiser in run mode

When the visualiser is open the user enters a program into the code window and selects run, triggering the events in Figure M.3. VisualiserCtrl creates an instance of class Lexer and passes to the lexer the contents of the program code window. It then calls the method makeTokens() to create a List object which holds the complete list of tokens. Any errors are sent to the static class BugCatcher for display within the VisualiserCtrl.

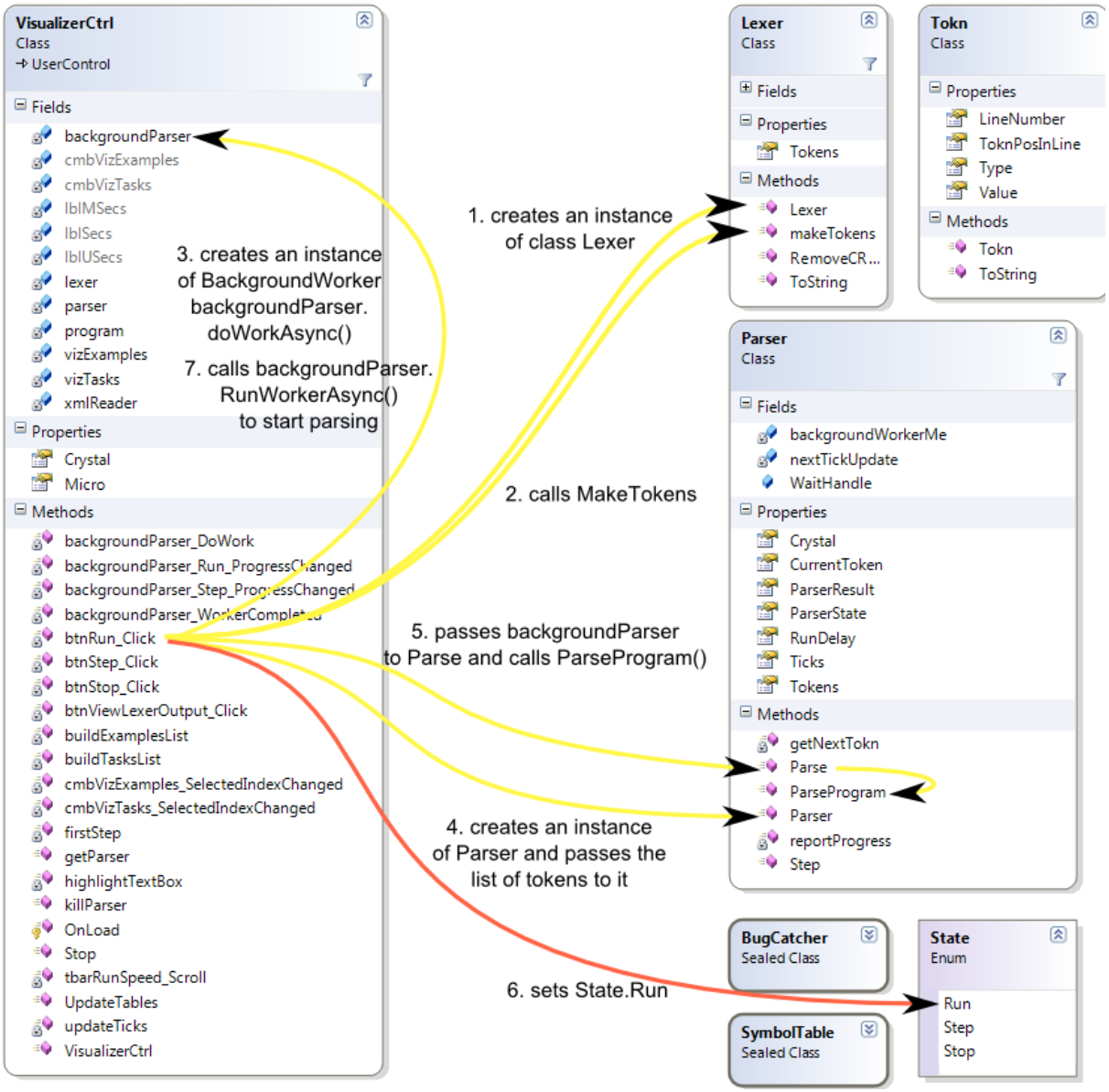


Figure M.3: Visualiser run mode initial states and background worker thread

A key aspect of development of software for Microsoft Windows is to make it as responsive as possible to user input; a GUI that blocks user input whilst some part of its process is completing, e.g. opening a file or in this case parsing a program, is unsatisfactory for user experience (Microsoft Corporation, 2013e). In general this is achieved by not running large blocks of code in the same thread (context) of program execution as the GUI but in a thread of its own. Thread

use in the .Net Framework comes in a range of implementations: creating your own thread; using a thread from the ThreadPool; implementing a BackgroundWorker (Microsoft Corporation, 2013a) , which uses a thread from the ThreadPool or the newer Task class (.Net Framework 4.5). BackgroundWorker was chosen for a number of reasons: System Designer has been built to date using .Net4 and it would require existing computer equipment in the classroom to be updated to make use of .Net 4.5. The added features of Tasks are not required for a modest case using just one thread; it is still a recommended solution by Microsoft (Microsoft Corporation, 2013h), and the reporting of progress to the foreground thread is more straightforward using a BackgroundWorker (Cleary, 2010).

Another control mechanism used is that of a flag using the enumeration 'State', which is used to regulate the parser's internal processes.

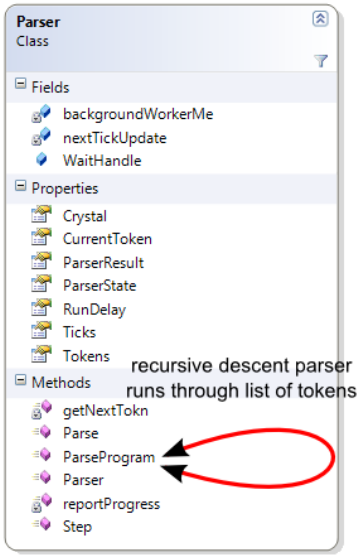


Figure M.4: Token recursive processing

The parser is a recursive descent parser (Figure M.4) and will recursively parse all tokens passed to it until they are all parsed.

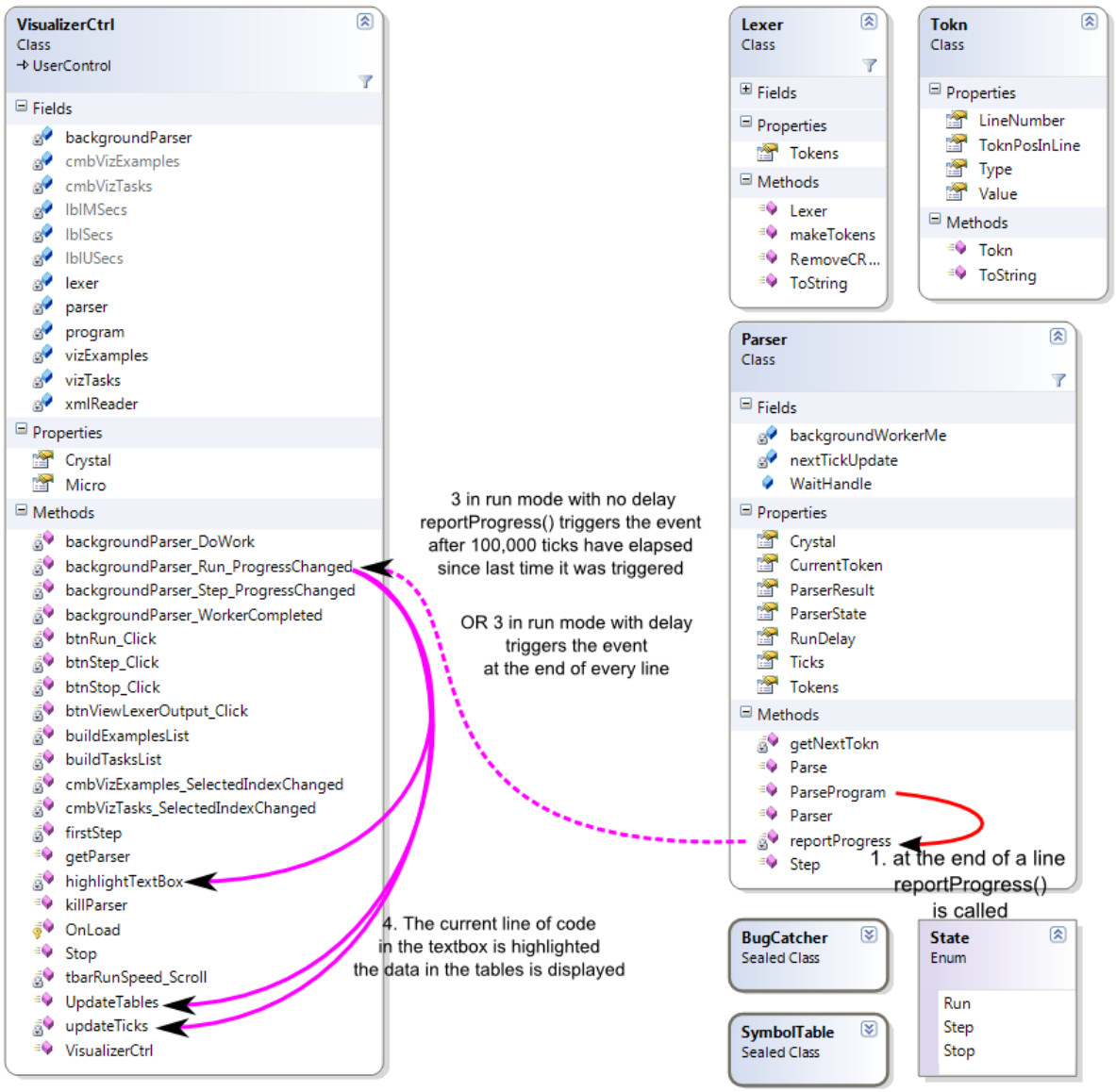


Figure M.5: Visualiser simulation of microcontroller timing

As the parser runs it keeps track of the run length of the program. A variable 'ClockTicks' is used to simulate a counter of the clock cycles taken to execute the commands. As this is an interpreter and not a compiler, actual counts of assembly code cannot be used; so an estimate of the number of cycles taken for known instructions is used instead. Being a RISC (reduced instruction set computing) processor with most instructions being executed in single clock cycles, these are more easily accounted for than in a CISC (complex instruction set computing) processor. The Ticks counter is incremented for most tokens encountered by the parser. As well as this any time related activities are accounted for using the Crystal value from the block diagram (e.g. during delay functions a 1000mS delay would be 1,000,000 Ticks at 1MHz and 8,000,000 Ticks at 8MHz).

Whilst the parser is parsing tokens it must communicate with the foreground thread at particular times. The first of those is when a carriage return, or end of line (Figure M.5), is encountered in the list of tokens (this is represented as the tokens tLF and tCR). The BackgroundWorker class has a useful feature for communicating back to the calling thread called ReportProgress. On encountering a tCR token the method reportProgress() determines what state the parser is in, either the Run or Step state, and if in Run state what the VisualiserCtrl delay value is set to. If in Run state and there is a delay of zero between the parsing of each line of code then backgroundWorker.ReportProgress() is only called every 100,000 ticks. This is done to reduce the load on the GUI from what may otherwise be too many events in a short period of time. During state Run if the parser has a non-zero delay value set (it can be changed from 0mS to 2000mS in increments of 100mS by the user) then the method reportProgress() will call backgroundWorker.ReportProgress() at the end of every line.

During parsing the parser will declare and access variables in memory, catch errors, find and use symbols, constants and function calls. Rather than one symbol table for all these items separate ones are used to handle the different requirements of each. Symbol processing is described in Figure M.6.

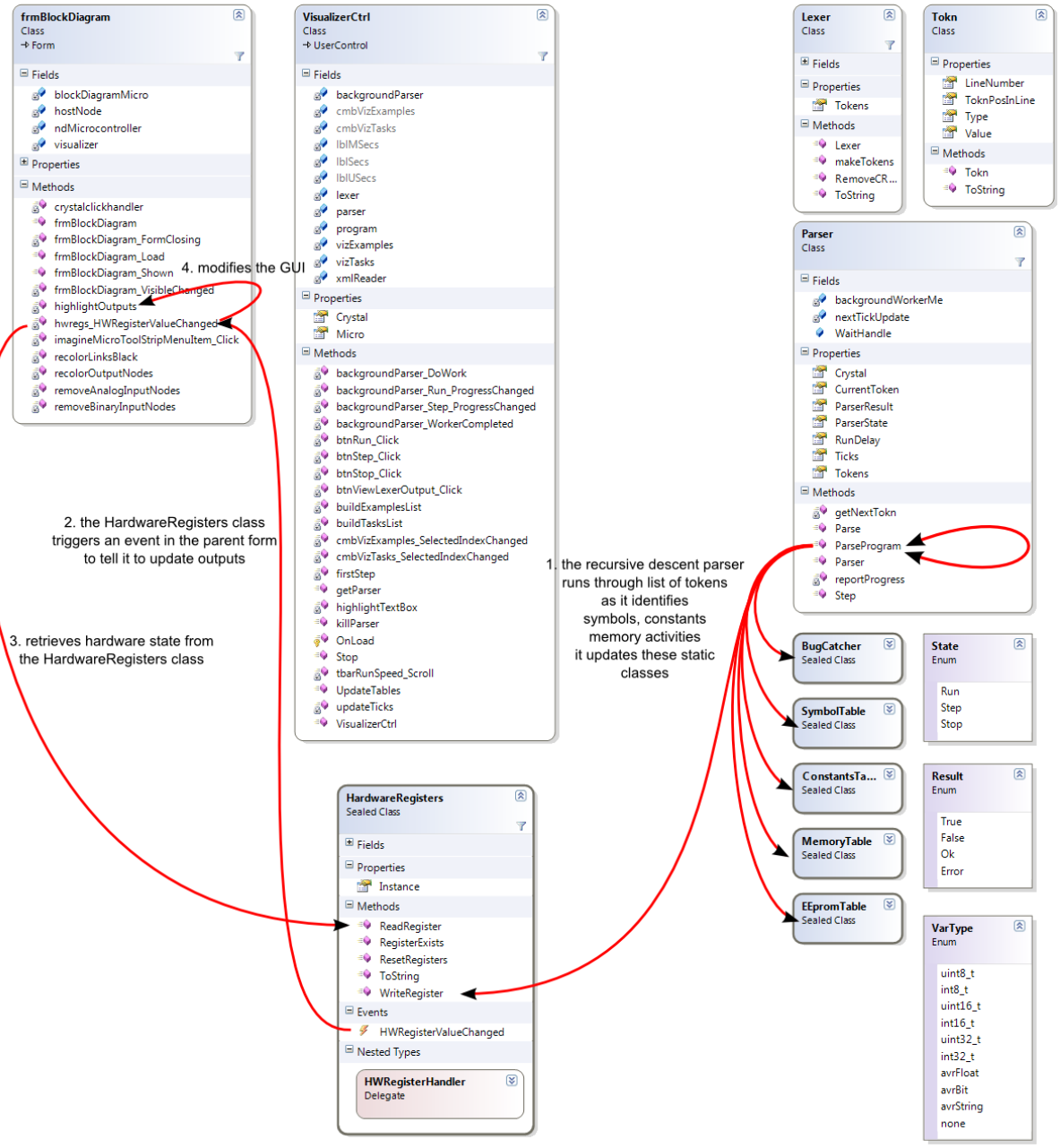


Figure M.6: Visualiser capturing symbols

The storage mechanism of an ObservableConcurrentDictionary was used for each table; these types were chosen because they easily facilitate bindings with the visualiser class which is a WPF usercontrol.

- class ConstantsTable – ObservableConcurrentDictionary

- class FunctionsTable – ObservableConcurrentDictionary
- class SymbolTable – ObservableConcurrentDictionary
- class MemoryTable – two off ObservableConcurrentDictionary (Memory and SRAM)
- class EepromTable – not implement yet
- class BugCatcher – ObservableConcurrentDictionary
- class Registers – ObservableConcurrentDictionary

The classes all use a ConcurrentDictionary rather than a Dictionary as it will be accessed by both threads concurrently. A Dictionary is used as it approaches an O(1) operation (Microsoft Corporation, 2013b) time as opposed to a List with its linear O(n) time complexity. ConcurrentDictionary has added complications over a standard Dictionary in terms of the locking and unlocking required to make it thread safe; the effect of this is not significant however ("C# Dictionary vs. List," 2013).

The parser in the BackgroundWorker thread will read, calculate and write values to the input and output registers (PORT/PIN) and at the same time the GUI in the foreground thread will read output PORT value changes and write input PIN changes. Thread safety is a crucial aspect of multi-threaded applications where more than one thread will add or update a collection concurrently.

A second consideration is the manipulation of the GUI via the BackgroundWorker thread. This is a cross thread operation as explained in Figure M.7. The red sections refer to code running in the foreground thread and the blue sections of code refer to the code running in the BackgroundWorker thread. When the HardwareRegisters are updated by the parser, the event fired runs in the BackgroundWorker thread not the thread of the class it belongs to. The GUI controls are running in the foreground thread and the .Net Framework detects such unsafe calls. The Invoke method supports cross thread access (Microsoft Corporation, 2013c).

Figure M.7: Thread safe operation between parser and GUI threads

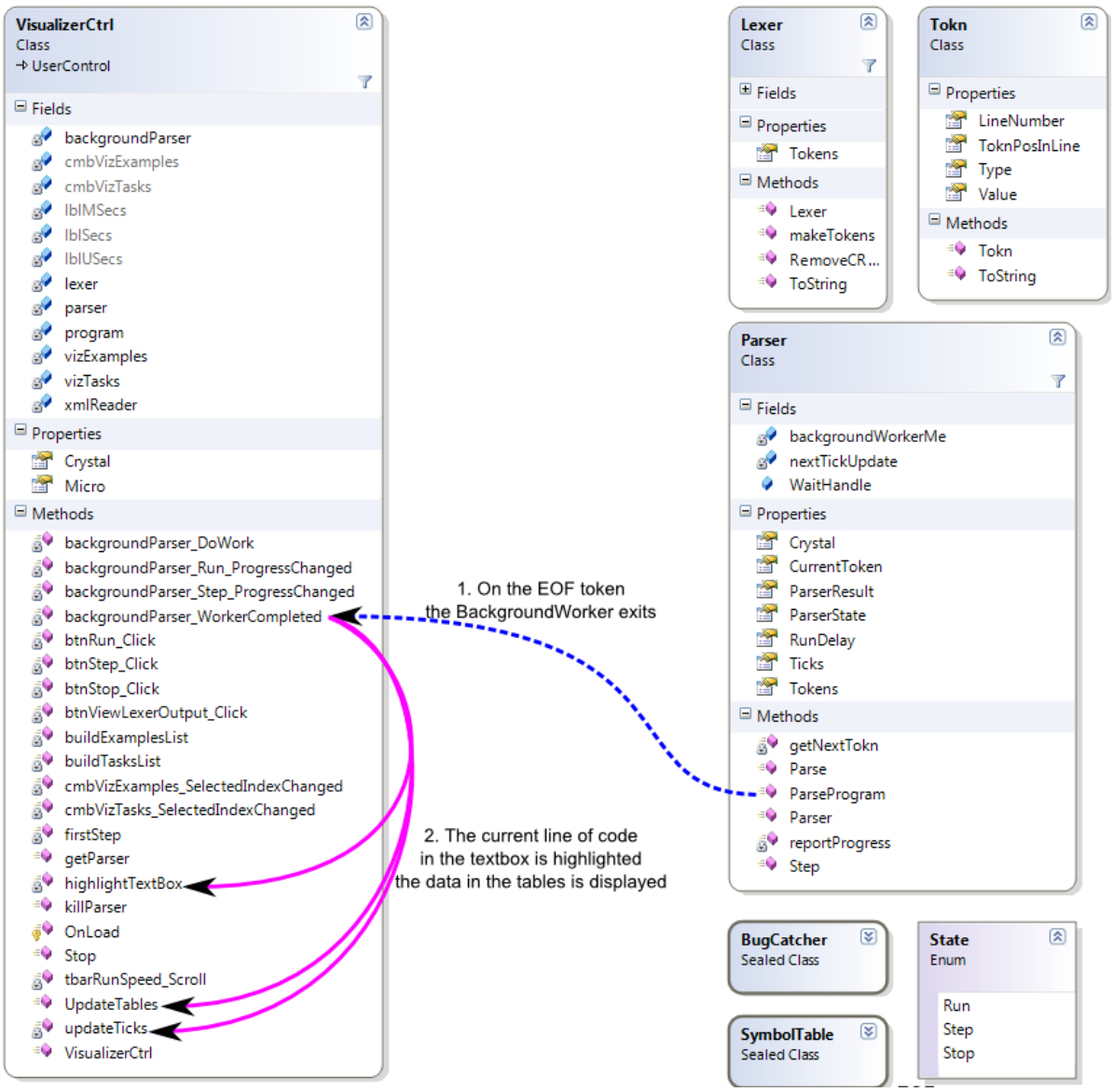


Figure M.8: Visualiser end of program

When the parser reaches the end of the program code it finds an end of file token (tEOF) as in Figure M.8. At this stage the BackgroundWorker exits and issues a signal and the VisualiserCtrl method backgroundParser_WorkerCompleted() is called which updates the GUI.

The final aspect of the parser Run state is being able to stop the parser in a thread safe manner without it locking up (Figure M.9). The BackgroundWorker includes a mechanism to avoid having to use Thead.Abort() which is an unsafe way to terminate threads (Microsoft Corporation, 2013f). The preferred method is to signal from the foreground thread to BackgroundWorker via the BackgroundWorker.CancelAsync() method (Microsoft Corporation, 2013d). This mechanism does not shut down the thread; it is a thread-safe way of sending a flag to the thread that it should exit itself. This requires some extra code in the parser as the BackgroundWorker.CancellationPending property must be checked regularly. In recursive code such as in a parser there are a number of while loops where the parser could become trapped and not see the CancellationPending flag. To avoid this all while loops have had added a check for this flag and will exit when it is found.

For instance in the method BitwiseShiftOperation the while method is:

```
while (  (CurrentToken.Type == TokenType.tShiftLeft || CurrentToken.Type == TokenType.tShiftRight) &&
!backgroundWorkerMe.CancellationPending && tokenIndex < Tokens.Count )
```

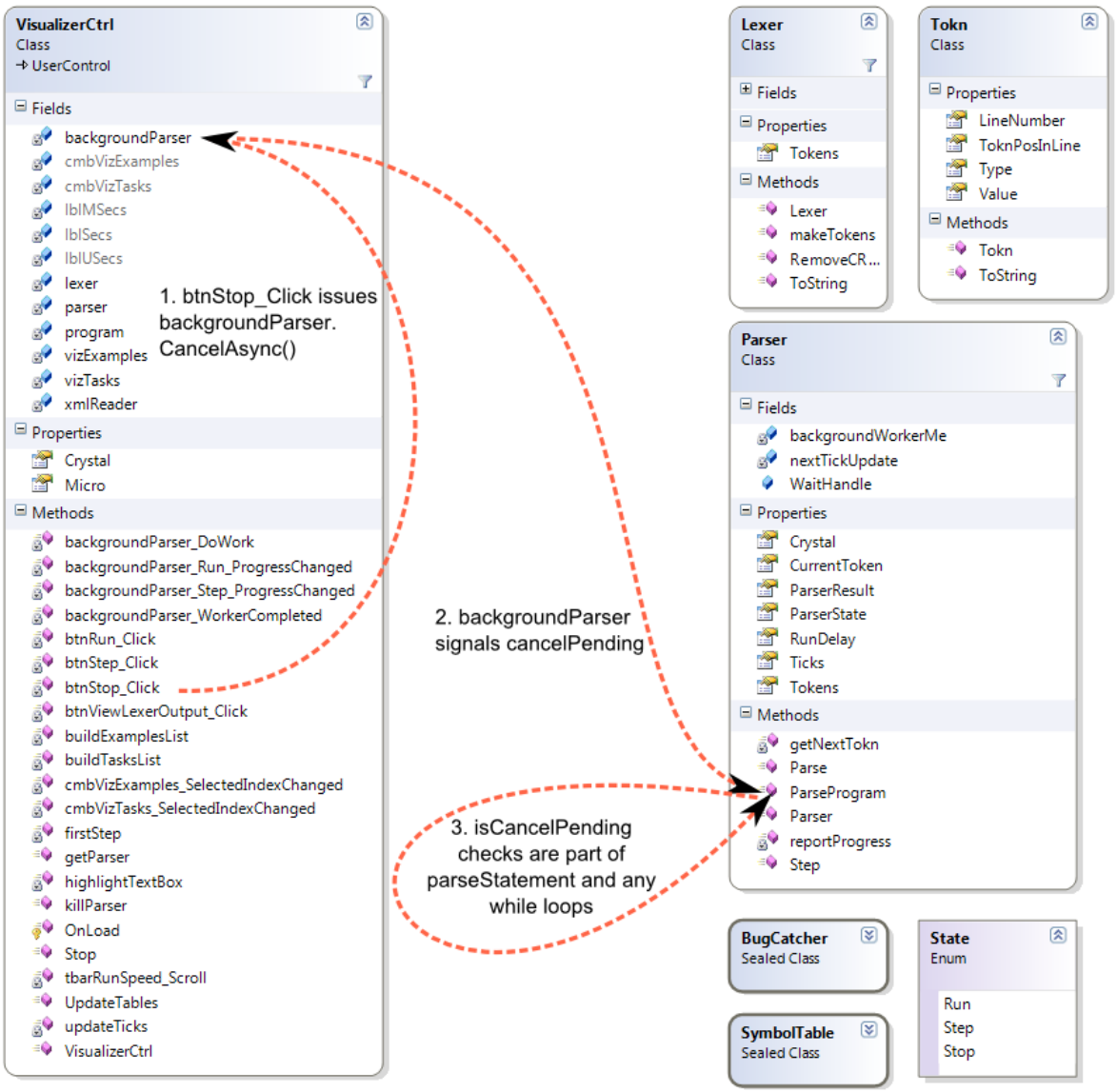


Figure M.9: Thread safe termination of the visualiser

Single Step

The second parser state is the ability for the user to single step through the program code.

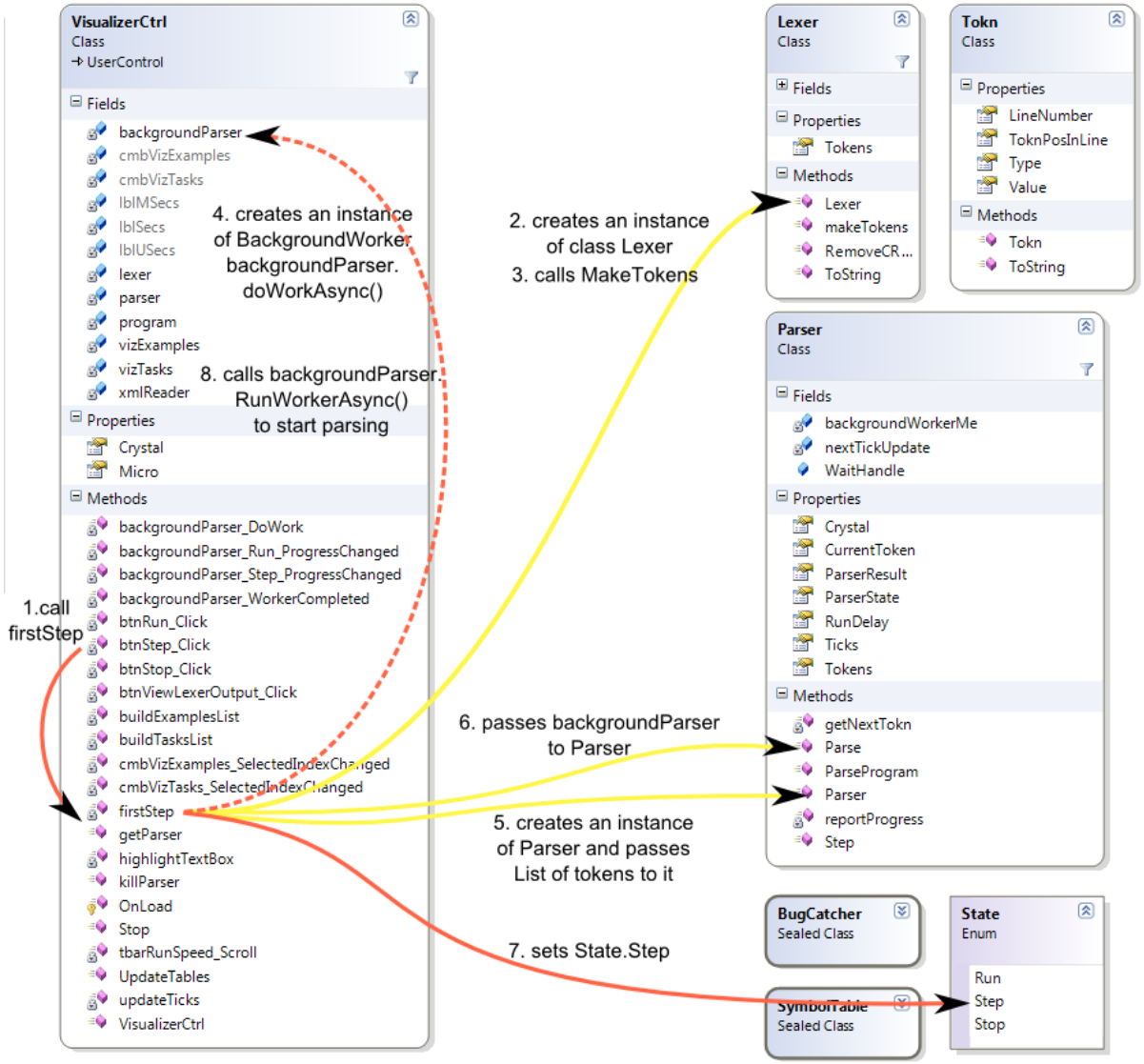


Figure M.10: Step mode initial state and background worker thread

In the Step state (Figure M.10) the VisualiserCtrl requires the parser to be instantiated on its first click of the step button and after that to issue step commands on subsequent presses of the step button. The process started by the call to the method firstStep() is different only in respect of setting the state of the Parser.

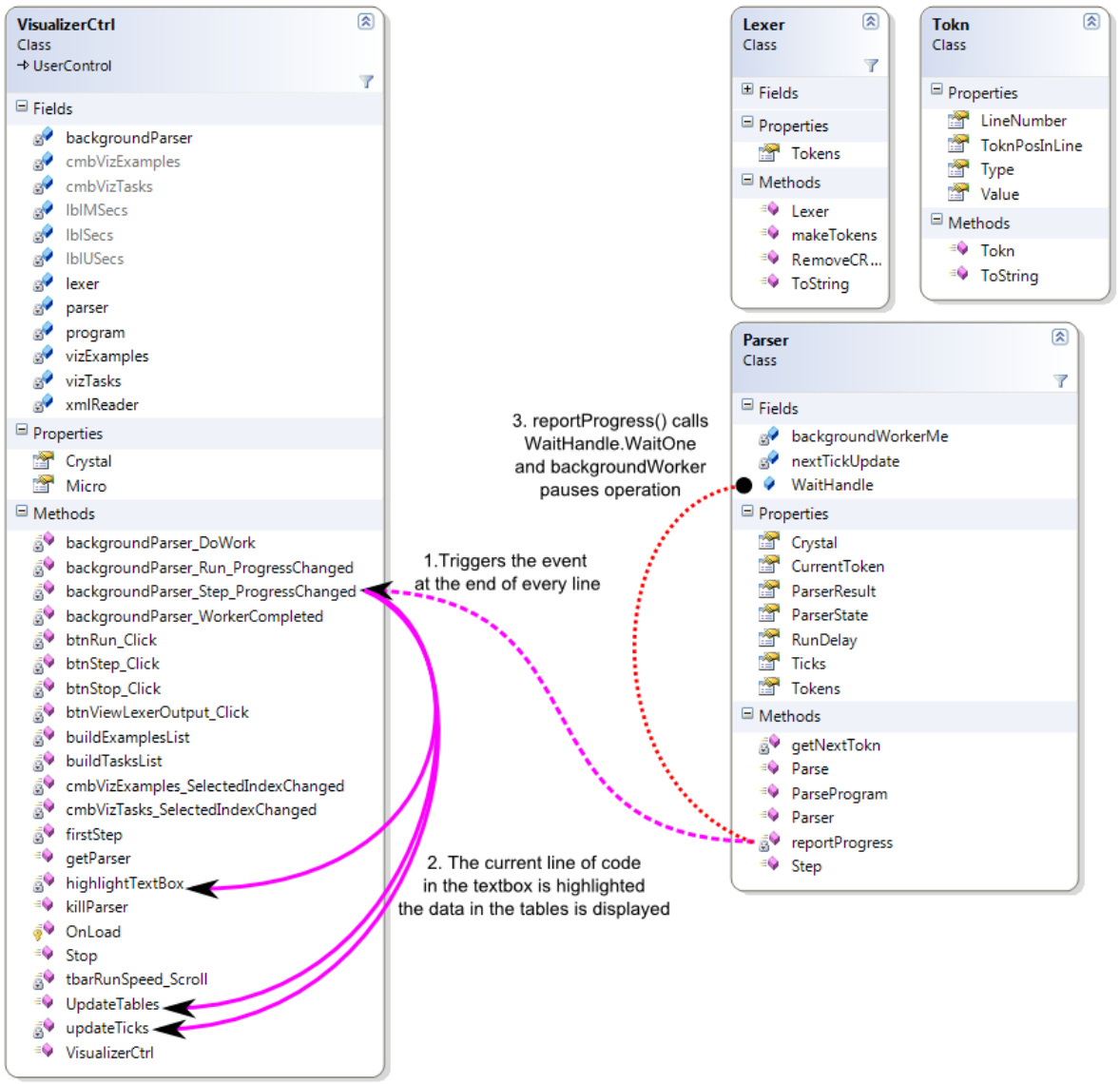


Figure M.11: WaitHandle use in visualiser background worker thread

The process of parsing is the same as in the Run state however the thread must be paused and resumed at the end of each line by the user from the step button on the GUI (Figure M.11). The use of Thread.Suspend and Thread.Resume to synchronize the BackgroundWorker is not recommended (Microsoft Corporation, 2013f); the preferred method is to use a WaitHandle AutoResetEvent (Microsoft Corporation, 2013g).

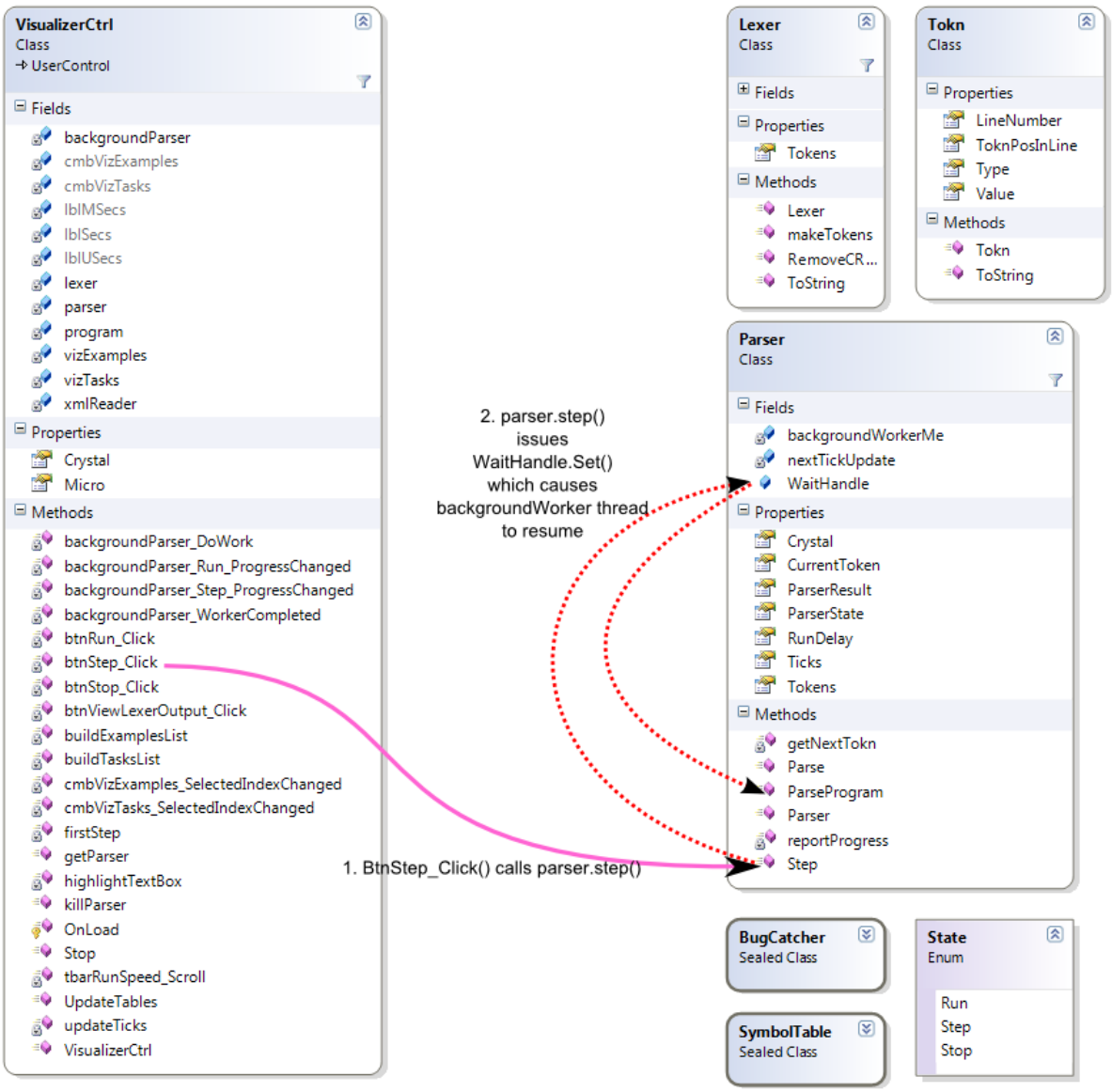


Figure M.12 User clicks step to release WaitHandle

The BackgroundWorker will suspend execution until it receives a WaitHandle.Set() call via the step button on the GUI in the foreground thread (Figure M.12).

Finally the end of file and stop actions work as per the previous descriptions in the section on running program code.

.

Appendix N: The context of learning in embedded systems at Mount Roskill Grammar School

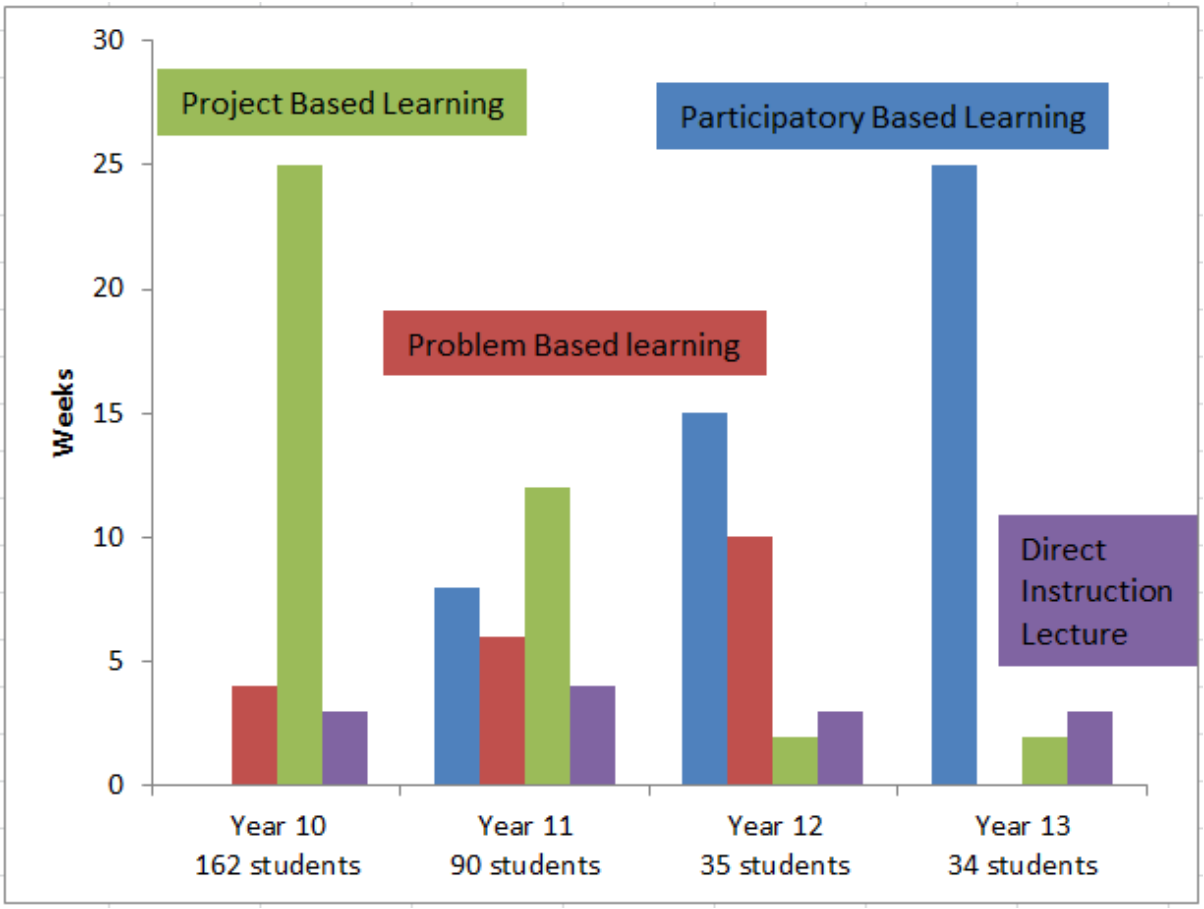


Figure N.1: MRGS Electronics: student time in different learning tasks (2012 data)

The programme of learning has been developed over time to make the most of the time allocated to a single course of learning in secondary school and the curriculum. Figure N.1 shows the transition from predominantly project based learning (non-contextualised) to problem based learning (contextualised) and participatory based learning (community oriented). The relatively small amount of direct instruction classes are spread throughout the course and confined to a limited range of analog and digital electronics theory. There is no analog theory beyond Ohm's laws applied to voltage dividers and current limiting resistors. Digital theory is limited to microcontrollers and no other digital logic is covered. This theory is presented as required at strategic times during the courses.

Practical tasks as part of project based learning build the desired skills required in the course, such as soldering, PCB layout and design, interfacing and programming. Project based learning tasks are also focussed on developing in students a different way of learning to that they often experience at secondary school. This involves a preeminent focus on building conceptual understandings using real world phenomena rather than book exercises. This develops curiosity and intrinsic motivation for learning and is a precursor to the deeper problem based learning.

Problem based learning increasingly features in the courses at years 10, 11 and 12. This extends project based tasks, as its goals are centred on students developing self-responsibility and metacognitive abilities as they undertake personally guided research, trialling and testing of their projects.

Participatory or socio-cultural technological practice is a growing focus of the learning from years 11 to year 13. In Year 13 students are required to find a unique client and work in conjunction with them to solve some issue. This is the most demanding year level to manage.

Hardware

Microcontrollers at all year levels come from the ATMEL AVR range and I have designed development PCBs specifically for each year level. The PCBs at year 11/12/13 are designed to piggy-back with LCDs and have prototyping areas for students to work on interfacing various IO devices. Students do learn PCB design but produce smaller boards for their assessments; however a small number of students will design their own PCBs for their projects.

Year 10

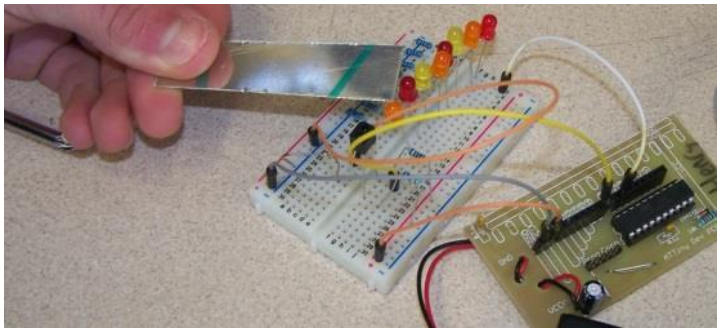


Figure N.2: Year 10 development board with IR reflective detector and LEDs

The emphasis is on building sound practical skills and motivating the students to want to continue in the course. The year 10 development board (Figure N.2) uses an ATTiny461. It has the

fewest connections of all the development boards and is used with a breadboard for experimenting.

There are two courses offered at year 10, a single semester and a double semester / full year. All students undertake a microcontroller based project which is a sign or festival oriented display using multiple LEDs and one input sensor. The full year students also undertake an audio amplifier design project.

Year 11

Students undertake a digital clock design project. The development board is currently based upon an ATtiny461 microcontroller (Figure N.3). It includes a small section of prototyping area for students own interfacing work.

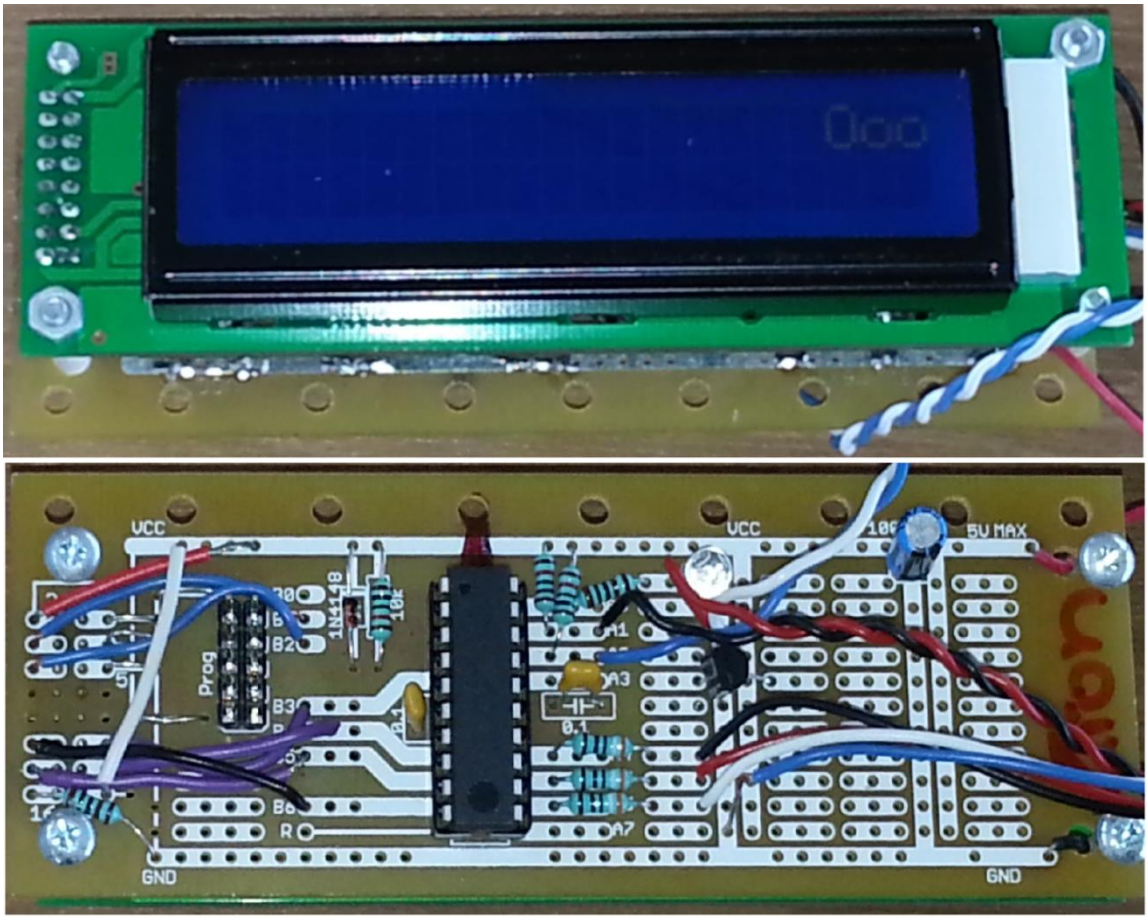


Figure N.3: Year 11 development board

Year 12

Students undertake a project of their own choice which is aimed at monitoring and collecting data so as to improve the performance of something. The development board is designed to piggy back with a 20x4 character LCD (Figure N.4).

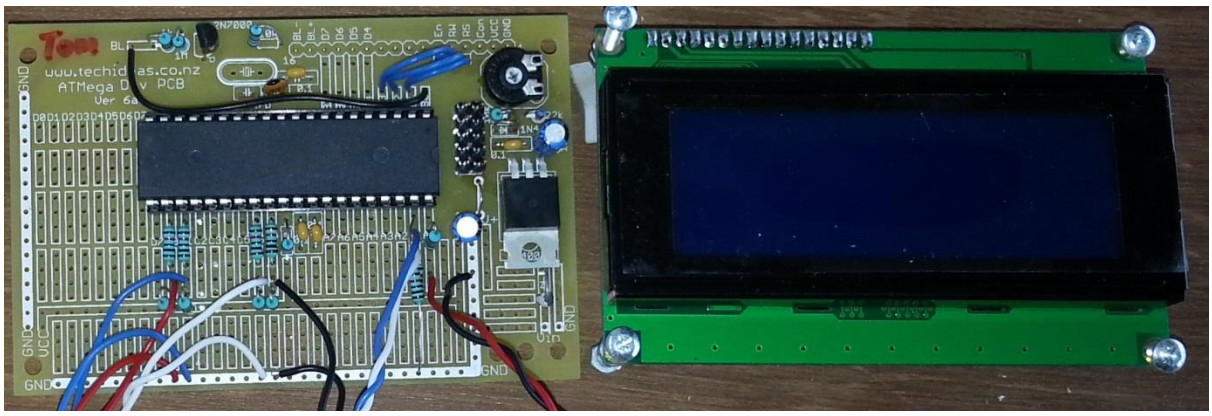


Figure N.4: Year 12 development board

Year 13

Each student in the class must find a client and work with them to solve some issue. All students have the same development PCB (Figure N.5) and a 128*64 monochrome graphics LCD. While most students will use this in their project some will design their own.

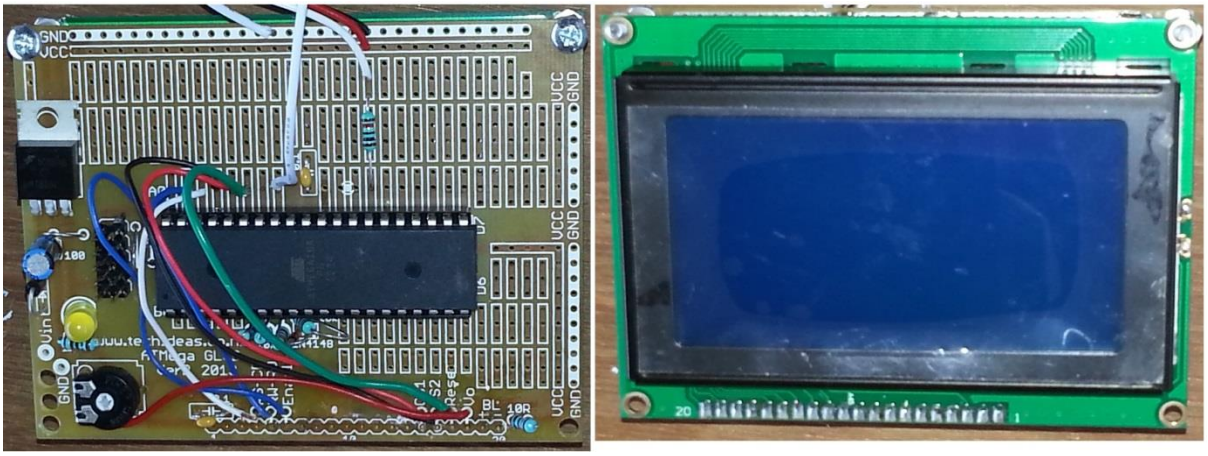


Figure N.5: Year 13 development board

Exemplars of year13 projects include: remote controlled lawn mower, tennis scoring and remote display, car ventilation system, vehicle distance sensing, home telephone exchange, American football thrower, ultrasonic ‘headlight’ for a blind person, fencing trainer, badminton trainer. These and other projects may be found on the course website at http://dl.dropboxusercontent.com/u/144296997/journals.html.