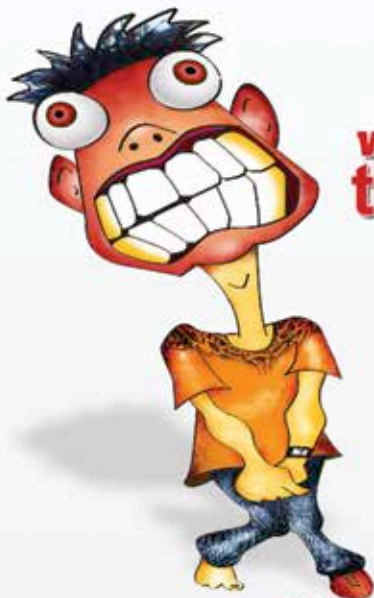


To **EMBEDDED SYSTEMS**

- What are Embedded Systems?
- Importance, usage and scope of Embedded systems
- Hardware architectures
- Hardware design and challenges
- Software systems
- Programming Embedded systems
- Future of embedded systems and career opportunities



**www.
thinkdigit/forum**

Join the forum to
express your views
and resolve your
differences in a more
civilised way.

**thinkdigit
FORUM**

Post your queries
and get instant
answers to all
your technology
related questions



One of the most active online technology forums
not only in India but world-wide

**JOIN
NOW**



www.thinkdigit.com



EMBEDDED SYSTEMS

powered by



CHAPTERS

EMBEDDED SYSTEMS

JUNE 2013

05

PAGE

What are Embedded Systems?

Ever wonder how everything works around you? From your washing machine to air traffic control systems, they all work on embedded systems.

13

PAGE

Applications of Embedded Technology

Here you will find out how traffic lights can be made intelligent, why shopping in the future is going to look like shop-lifting, and what keeps a pacemaker beating.

26

PAGE

Hardware architectures

Embedded systems work on hardware that is completely different from your PC. Get ready to enter the miniscule world of microcontrollers, sensors and components

CREDITS

The people behind this book

EDITORIAL Executive Editor

Robert Sovereign-Smith

Writers

Arpita Kapoor
Cyril V
Jait Dixit
Mohit Rangaraju
Vaibhav Kaushal

Features Editor

Siddharth Parwatay

Contributor

Copy Editor:
Infancia Cardozo

DESIGN

Sr. Creative Director
Jayan K Narayanan

Sr. Art Director

Anil VK

Asst. Art Directors

Atul Deshmukh
Anil T

Sr. Visualisers

Manav Sachdev
Shokeen Saifi

Visualiser

Baiju NV

Consulting Designer

Vijay Padaya

38
PAGE

Hardware design and challenges

Ensuring the quality and stability of the design is essential when it comes to embedded systems. Here we discuss some of the principles and challenges.

47
PAGE

Software systems

Understand internal modules of the system such as kernel, scheduling, IPC, memory management, I/O and time services in addition to operating systems prevalent in this area.

59
PAGE

Programming embedded systems

It's time to get your hands dirty in code. Find out about the lifecycle of an embedded project, hardware, cross-compiling and toolchains, debugging and emulation, RTOSes, tracing and math workbenches to be utilised when working with DSPs.

80
PAGE

Future and career opportunities

Embedded systems will give rise to the era of ubiquitous computing, context awareness and other intelligent systems. Welcome to a brave new world.

© 9.9 Mediaworx Pvt. Ltd.

Published by 9.9 Mediaworx

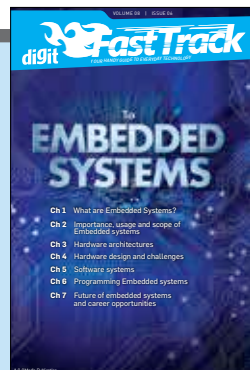
No part of this book may be reproduced, stored, or transmitted in any form or by any means without the prior written permission of the publisher.

June 2013

Free with Digit. If you have paid to buy this Fast Track from any source other than 9.9 Mediaworx Pvt. Ltd., please write to editor@thinkdigit.com with details

Custom publishing

If you want us to create a customised Fast Track for you in order to demystify technology for your community, employees or students contact editor@thinkdigit.com



COVER DESIGN: PETERSON P

Introduction

The word 'computer' usually conjures up images of a desktop, a laptop, a server or perhaps if you're a really big thinker, even a mainframe. In reality computers are not just dumb boxes which just sit around on tables. As a matter of fact, computers are embedded into everything ranging from your microwave to the Boeing 777, your smartphone to the Curiosity Mars Rover. All in all we're surrounded on all sides by these small wonders; about 90% of the processors manufactured across the world are powering what are also called embedded systems. Such systems despite being small and seemingly non-existent are the reason for the rapid growth and advancement in other areas of technology.

These devices are responsible for powering everything from mundane home appliances, traffic lights and simple robots to bewilderingly complex systems deployed for process and production control at automated manufacturing plants, power stations and telecommunication systems. They're also used heavily in avionics and air traffic control. Apart from this, embedded systems find use in space applications – Space X Dragon, Mars Pathfinder and Curiosity all carried such systems.

This FastTrack will delve into the workings of such Embedded Systems. We'll start with an introduction to embedded systems and why are they important followed by their functions and the scope of their applications. Then we'll take up, hardware – design principles, layout considerations and component selection along with the challenges that such systems throw at designers due to constraints on power consumption, heat dissipation and real time performance. On the software side we'll talk about how software is written for such systems and how it differs from non-embedded software. We'll also discuss the operating systems employed on such devices and their functioning. Finally we'll talk about the future with mobile and ubiquitous computing, how will we interact with computers which are embedded right into our environment and how will the revolutionize the way in which we interact with it.

Of course there's no way this little booklet will be able to cover every aspect of such systems in detail but we've tried to give you a birds eye view of how such systems function and hopefully spark your curiosity and spur you onwards into delving deeper into this micro-universe. We hope you enjoy reading this FastTrack as much as we enjoyed writing it. As always, happy reading.

WHAT ARE EMBEDDED SYSTEMS

How does your washing machine know when to stop spinning and start rinsing? How does a fridge realise that the door is open and hence turns on the light? How does your car know that you are going to crash into that wall behind you?

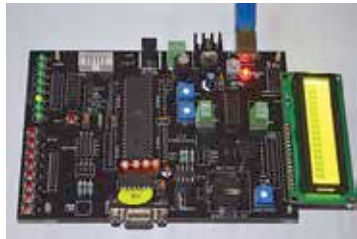
All these functions are carried out by the help of embedded systems. Why should you know about them? Well, normally users frolic in the joy that these technologies bring without ever understanding their intricacies. Don't you want to stay one step ahead of the rest? Also, as cited before, your home is probably full of them, and wouldn't it be a good idea to understand these little pixies of the electronics world?

Remember greeting cards? Actual ones, not those pathetic e-versions. Now remember the musical ones? The music starting to play when you opened the card was powered by a simple 4-bit embedded system. More complex 128-bit embedded systems are used in nuclear power plants, satellites, aircraft guidance systems because, well, these systems are more complex.

So what is an embedded system?

An embedded system is a microprocessor-based device which has an inbuilt operating system. This OS is designed to perform specific functions, that too under a time constraint. For example, take a standard (dumb) mobile phone. You don't just want to call on it.

You would like to be able to reduce the screen brightness, to be able to adjust the zoom of the camera, or you may want to use it as a calculator. To bring about such variations in its functionality, embedded processors come to the rescue. Each embedded processor will handle one aspect such as screen brightness, or FM radio, and so these



Li2_Board_LCD

and all the other embedded processors for each specific function will collectively be called an embedded system. Moreover, it is working under a time constraint as well – you don't want to call up a friend and have to wait for ten minutes to hear what they are saying. This is called working in 'Real time'. A situation like that where you have to wait for more time before talking is not going to cause a life-threatening catastrophe (at least we guess that it won't) and so this system is called a soft real-time system. But if you consider, say, the working of a pacemaker (a small device that's placed in the chest or abdomen to help control abnormal heart rhythms – you can find the working in the next chapter), it just cannot afford to have even the tiniest delays in execution, and so a system like this is called a hard real-time system.

Thus, embedded systems are generally systems within systems, hence they generally cannot function on their own. Consider a Digital Set Top Box (DST) which might be sitting in your home. The A/V decoder i.e. the digital audio/video decoding system is an embedded system and is a very important part of the DST. Taking a single multimedia stream as input, the A/V decoder produces sound and video as output. But these signals received from the satellite by the DST contain multiple streams, which the A/V decoder doesn't like, and so comes another embedded system to save the day – the transport stream decoder. This decoder takes the incoming multimedia streams and splits it into separate channels and gives only the required channel to the A/V decoder. And so you have your fully functioning DST, all thanks to embedded systems.

Now we did say that embedded systems generally cannot or don't function on their own, but there are exceptions. These are stand-alone embedded systems. They take input, process them, and produce the desired output. The input here can be commands from a person, such as pressing of a switch, or they could be electrical signals from transducers. This output will then be used to drive another system such as the display on a microwave oven. A network router is a stand-alone embedded system. Built using a specialised communications processor, memory, a number of network access, and network ports, the network router routes packets from one port to another based on a programming algorithm, and hence is a stand-alone embedded system.

So can a PC that is only used for browsing be called an embedded system? Well, to be able to understand this we need to be able to truly understand what an embedded system is considering its characteristics and various perspectives, so we'll get to that in a minute. First, let us see how a computer is different from an embedded system.

First, what is a computer?

Now, we know that you may know this already, but bear with us, there is a point to this. A computer is a system that will have the following or more components:

- A microprocessor
- A large memory comprising the following two types:
 - Primary Memory (or semiconductor memories – RAM, ROM, and fast accessible caches)

- Secondary Memory (hard disks, optical memory in DVD-ROM, using which different user programs can be loaded into the Primary memory and subsequently run.
- Input/output Units such as touch screen, modem, etc.
- Input units such as the mouse, keyboard, scanner
- The Operating System
- Networking units such as Ethernet Card, bus drivers, etc.
- User Interfaces and application software which are mostly in secondary memory



The innards of a PC

What is an embedded system

Now have a look at an embedded system. There are three main components of an embedded system.

- It embeds hardware so as to give computer-like functionalities to the device.
- It embeds main application software generally into a flash or ROM and so the application software performs concurrently the number of tasks.
- It embeds a Real Time Operating System, or an RTOS. The RTOS is basically a very, very efficient boss. It enables the execution of concurrent tasks. It also provides a modus operandi by which it allows the processor (who in this analogy is the employee) to run each process (or 'project') as per scheduling and to switch between various process depending on current priority levels. The RTOS, in effect, sets the framework of rules during the execution of application processes so as to enable finishing of a process in good time and with the appropriate priority.

Getting the picture? Embedded systems have also been defined as devices that include a programmable computer but by itself are not intended to serve as general purpose computers, or to elaborate further, the computer is hidden or embedded in the system. Any device that seems to be intelligent and is not a computer can generally be assumed to be an embedded system. The gate which opens automatically when a car is closing in on it? Embedded system. A valve which opens only when the humidity in the room is at a certain level? Embedded system. But then, we hear you wonder, can we have an embedded Operating System ON a PC?

Embedded OSes on a PC?

Embedded operating systems have been around for quite some time, like Symbian for phones and also Linux, and are basically operating systems with all the bells and whistles removed, and are capable of carrying out only a reduced number of functions.



An embedded system

Multimedia notebooks have the

option of playing a movie without having to load up the Operating System and lug your way to the DVD play option – this is an example of an embedded OS. To understand further, consider a mobile phone. When the phone is turned on an OS boots which handles all the interface and the services of the phone, and upon this additional programs can be loaded such as JAVA applications. So an embedded system can either be made specific to an electronic device, or it can be made specific to any general purpose operating system tweaked to run on that electronic device. For a computer, an embedded OS is an additional flash memory chip which is installed on a motherboard that can be accessed through booting the PC. But now you ask – “why does a PC need a separate hardware operating system since it is more than capable of using all of its features by itself?” Well, imagine you are in a class of forty and your professor wants Viresh (replace with whatever name you prefer) to go to the Chemistry Lab to complete some experiment he missed out on. Wouldn't it look silly if the rest of the students follow him to the lab and stare at the wall while he does what he has to? Similarly, if you all want to do is waste time on Facebook do you really need to have your computer boot up and access your hard drive and rather power hungry graphics card? Continuing the train of thought, wouldn't a class of forty take more time to reach the lab – slapping each other, getting lost, or doing whatever it is kid's do in school nowadays than just Viresh going alone (think boot-time)?

So yes of course you can have an embedded OS on a PC. If you just want to watch a movie you don't have to sit through the painfully long (and it's always painfully long) boot time. More importantly, if you are a travelling businessman or a journalist and you just have to check up on something online you can do so in one step. However, don't just run

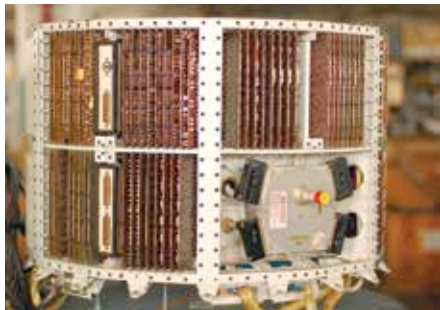
out and ask your nearest computer walla for an embedded OS on your PC/laptop. Do you really need one? Decide if you will really use it – else you'll be paying extra for something which has no function other than making your not-so-tech-savvy friends go “Wow!”.

So isn't any device from time immemorial an embedded system? Well, no. The first use of an embedded system was in 1961, and it was only in the 1980s when it was able to break into the market commercially. Before this, intelligent systems weren't possible as embedded systems weren't feasible, because of financial and logistical constraints. Read on to take a fast-track trip through the history of Embedded Systems and understand why they are such an important milestone in technological advances.

History of embedded systems

The minuteman ICBM

The Minuteman intercontinental ballistic missile was the first to use an embedded computing system, and was built in 1961 for the US Air Force. This embedded system was the Autonetics D-17 guidance computer. Embedded microprocessors, in addition to the advances in solid-fuel propellants enabled the Air Force to develop its first solid-fuel ICBM. The embedded system controlled avionics (a cool way of saying aviation and electronics in one word) and inertial guidance for the missile. The Minuteman was a 3 stage solid-propellant rocket-powered ICBM with a range of around 5,500 nautical miles. The system helped in creating a mass producible, simple and efficient ICBM capable of destroying enemy targets with consistent reliability, which is how wars are won.



The Autonetics D-17

Apollo Guidance Computer

The Apollo guidance computer was a digital computer introduced in August 1966 for the Apollo Program that was installed onboard each Apollo Command Module and Lunar Module. It was the first integrated

circuit-board computer and had a 16-bit word-length and provided computation and electronic interfaces for guidance, navigation, and control of the spacecraft. At the time of the project's inception, the AGC was considered to be the riskiest item in the Apollo Project as it was the initial stage of the use of the monolithic integrated circuits which reduced size and weight of the total apparatus.

F-14 Tomcat

The Central Air Data Computer (CADC) built into the F-14A Tomcat and launched in 1970 was the first microprocessor-based embedded system. It provided cockpit information such as altitude and air-speed and also handled avionics and flight handling computation of the Tomcat's swing-wing configuration. The CADC consists of 8

The F-14 "Tomcat"



microprocessors, 19 memory chips, and was designed by MOS-LSI. For guiding the aircraft control surface no cams or gears were used – everything was controlled by hydraulic systems and electric motors and so the CADC represented the first completely digital control system for guiding aircraft control surfaces. Details of the CADC only came out in 1998 when its lead designer published a paper. We think it's safe to assume that the lead designer lived a long, troubled life, being stalked night and day by armed, black-suit-wearing men called Agent Smith, who work for the "Government".

Modern usage

As you can see from the examples above, military and aviation applications drove embedded systems during its initial stages. Thankfully, from 1990 onwards things have changed for the better. The leading industries reaping the benefits of embedded systems have been the healthcare industry with integrated diagnostic devices, progressing to cellphones. The driving force behind most of these new applications of embedded systems is the availability of 32-bit processors and the fact that development tools are easier to use. These, in turn, have increased the usable computational power

enormously and at the same time with modern CPU manufacturing a lot of the electrical power constraints have fallen. Today, you wouldn't have a Kinect Controller for the Xbox 360 if it weren't for embedded systems!

The rise of embedded systems?

Due to systems-on-a-chip integrations which combine a microprocessor, RAM, I/O controllers and other control systems into one integrated circuit package, embedded systems were able to emerge into the marketplace in the 1980s. They were, of course, less flexible than conventional computers but they were also much cheaper. Unfortunately, due to power utilisation, electrical, and timing constraints, most of the embedded systems remained as 8-bit systems while the rest of the computing world moved on to 16-bit and 32-bit processing.

So now processors have shrunk in size along with an increase in performance; power consumption has been drastically reduced, and the cost of processors has fallen considerably to quite an affordable level. In addition, the design cycle and total development time has been reduced because of two reasons; there is now more awareness that rather than using a totally hardwired electronic system, if a programmable processor is incorporated in the circuit it makes the design more robust; and the concept of a development environment where the system can be prototyped and a simulation can also be carried out. Along with this, the development of Java and other standard run-time platforms, the emergence of integrated software environments which simplified development of the applications, and debatably the biggest of them all is the coming together of Embedded systems and the internet has contributed greatly to the rise of embedded systems. This makes the networking of many Embedded systems to operate as a part a system across networks like LAN, WAN, or the internet possible.

APPLICATIONS OF EMBEDDED TECHNOLOGY

Check out some of the cooler applications of Embedded Technology and see to what extent it is shaping the world into a better place to live in. Here you will find out how traffic lights can be made intelligent, why shopping in the future is going to look like shoplifting, and what keeps a pacemaker beating. At the end of this section is a quick run-through of applications of Embedded Technology in various industries.

Intelligent traffic light control

The Traffic Light Controllers (TLC) used now are based on microcontrollers and microprocessors. This TLC uses predefined hardware, and so can only function according to the program and does not have the intelligence to be flexible on a real-time basis. Ever had to wait at a crossroad when there is absolutely no vehicles incoming from the adjoining roads but still you have to wait for the timer to count-down to zero because someone thought that was a pretty neat idea? Ever found yourself philosophising about your life because there is nothing really else to do except inhale exhaust fumes? All this in addition to the fact that there is so much petrol being wasted for absolutely nothing. That horrid experience could soon never have to be experienced by anyone again. Now there has been a proposed system of Intelligent Traffic Light controller which can be used to give the system flexibility of modification on a real-time basis. This new method will use sensor networks coupled with embedded technology to act smarter. Here, first the total traffic on all the connected roads on a crossing will be noted. After considering this, the timings of the red and green signals (no one really cares about orange signals, it seems) will be appropriately adjusted so as to ensure maximum efficiency in the movement of the vehicles to prevent major traffic jams.

Automotive technologies

Embedded systems are contributing to a safer system of driving. Every automobile manufacturer which comes out now uses embedded technology to make their cars sweeter. So what can we expect from cars eventually? Embedded systems can regulate speeds, just in case you're prone to putting your foot down and trying to be a speed demon, and the car will try and maintain safe distance to the car in front of you, and in some cases even intelligently make decisions based on traffic conditions, weather (and hopefully how drunk you are). Imagine that, the end of drunk driving and road rage? Currently, embedded systems in automobiles are rugged, as they are usually made up of just a single chip. Because of this ruggedness, there is a much reduced chance of failure, and almost no possibility that the system will get busy with another process and hang. This also makes the whole system small enough to fit inside a car's nooks and crannies.

One of the more exciting applications of embedded technology is that soon you could see the first driverless car (insert that driverless car show joke here), although it wouldn't be that cool, or that lame depending on how

you look at it. Ford has already started work on the Adaptive Cruise Control (ACC) which, as we mentioned before, will allow vehicles to maintain safe distances from each other. With ACC, the driver only has to set the speed of his car and the distance he desires to maintain between other vehicles and his car, and if he chooses to he can always over-ride the system by applying the brakes. A microwave radar unit or lasers transceivers fixed in front of the car determine distance and the relative speed of neighboring vehicles. The brakes and the throttle of the car is also handled by the ACC. Regular cruise control systems only track the desired vehicle speed, but ACC is unique in the aspect that it understands and intelligently adapts itself to the behavior of the respective vehicles and using Radar or Lidar it follows what we shall call the 'leader' vehicle at a time gap requested by the driver. When this 'leader' vehicle is not present, the ACC reverts to regular cruise control and defaults to the speed set by the driver. The Nissan Q45 and FX45, the Mercedes S-class, the Lexus 330 and 430, the Audi A8, and select Jaguar and Cadillac models all use ACC and they obtain the required data using forward-looking sensors.

However, ACC also has its drawbacks. It is vulnerable to interference, noise, and false alarms as it isn't yet a perfected system and because of this heavy delays may creep into the system because of which the vehicles ability to follow required vehicle and maintain required speed are dampened. To better this system, there is another version of the ACC called Cooperative ACC. In the CACC, the forward-looking sensors we mentioned before are now coupled with a wireless communication link that connects the leader and the follower vehicle and hence is able to pass information between the two so as to get a better judgement on the next decision which could be increasing speed or decreasing speed or applying the brakes. If this system is commonplace, all vehicles will be connected through this system and so traffic will adjust itself in the most beneficial way for the good of all the vehicles and hence will make our commutes a lot less frustrating.

In addition, the internet is paving the way for enhanced use of embedded technologies. Imagine driving past a petrol pump on your way to Lonavala only to be given a notification that you need a refill (which is pretty awesome), or crossing a toll naka and having the toll amount directly deducted from your bank account (which we just realised is not so awesome). This system is already in place in quite a few parts of the world.

RFID

While it is not technically an application of embedded systems, the use of RFID is going to be boosted quite a bit as it now RFID technology has reached the potential to now become an integral part in embedded-system design. Traditionally used in inventory management, developments in RFID technology along with high-speed, long-range readers now allow embedded-system designers to incorporate with much ease features such as simplified payments, counterfeit prevention, medical authentication, access control, dynamic pricing, remote asset tracking, and product histories. By incorporating the reader in an even deeper level within the product or system, Embedded RFID applications add local data-gathering features which serve to improve greatly the originally slated functions of the product. Embedded-RFID applications are found in a lot of countries in restaurants, prisons, shopping outlets, casinos, toll roads, factories, and vehicles.

One of the initial uses of RFID was by the US military (who seem to have always have been on top of every technology out now) during World War II. It was used to distinguish between enemy and friendly fire aircraft. By the 1970s and through the 1980s it was commercially applied to be able to track and identify items within a location. The problem here was that there was no standardisation of the software- with each developer presumably looking to get one-up over the other by using their own unique methods of communications. Because of this loss of brotherhood among these developers, the industry fell apart. There was very slow adoption, probably because everyone was wasting their time trying to figure out the nuances of every new RFID communication device with its unique signature, and because of this RFID was just considered to be just another one of those new technologies which was hyped up so much that when it

fell all people could do was to feel embarrassed they were ever part of it. Today, though, things have changed for the better. Developers realised where they had gone wrong in the past, and have been working on getting RFID back to the pedestal it once was on.



The Lexus RX330 had problems with the embedded anti-lock brake system

Now, a dedicated reader is an essential part of the design of an embedded-RFID application to interpret local tags. To better understand this, let us consider a vending machine. It's built-in reader can accept contactless payments from RFID cards, and can tag the dispensed items. Because these items have been tagged, the machine can intelligently keep a track of the contents of its inventory and can automatically order refills in case it is going to run out.

Embedded-RFID technology can also replace dangerous men in suits who throw you out of the casino after giving you a good thrashing when they find out that you have been cheating. Generally, cheating and sleight-of-hand movements when done by someone who considers himself an artist at it are difficult to detect, by human beings alone that is. With the appropriate set of data-capture tools, casino operators can sneakily monitor the behaviour of a player to detect if they are guilty of card-counting, can adjust promotions, and can minimise dealer errors which stack up huge losses for the casino. Seems far-fetched? We dare you to say that to International Game Technology and Progressive Gaming International. These two have jointly developed the table iD table-game-automation system which uses the combination of a series of RFID-chip-scanner modules and a software-based table manager. So how it works is that during play, chip readers at each position identify and also record each players bets. The iD system then studies players' betting patterns, records dealer activity, and assesses once an hour with a keen eye player's decisions. Information such as average bet and win/loss record is automatically updated and requires no user interaction.

The legend of RFID goes that you may enter a shopping mall, fill out your bag with whatever you have, and walk out without having to suffer the hassle of physically standing somewhere and paying. The RFID readers would simply scan the items and automatically deduct the required the amount from your bank account. While this definitely raises some questions, like for example –what if you place the item back into the shop-would you get back the money which was just deducted



Embedded systems powered driving?

from your bank? Or what exactly would be the procedure if you do not have the required amount for the items? While we ponder these questions, we hope embedded-system designers are working on these things

right now because as skeptical as we are if it works then standing in line in a supermarket would be talked about with an air of melancholy like our beloved cassettes.

Medical technologies

Medical technology has a myriad of slots which embedded systems can fit into. Take for example the pills that are equipped with 'smart' processors which assess the situation and accordingly repair organs or process information about cell formation or anomalies in cell operation. But that is in the near future – right now we have a system which has saved the lives of over two million people – the pacemaker-and it is all thanks to embedded systems.

A little history first.

In the 18th Century, it was already common information that electrical stimulation causes the muscles to contract and Charles Kite was the first to describe this idea, which is so familiar to us right now, of using electrical discharges to save people who have suffered a heart attack.

Based on this, the first pacemakers were built – but they were not exactly the ideal elixir to an ailing heart. They were large, bulky and ugly and were built around vacuum tubes and were powered by external power supply. Because of this design, it was impossible to be implanted inside a person.

In 1947, the transistor was invented. This ensured a very bright possibility for implantable devices. In 1958, Arne

Larsson received the first fully implantable pacemaker. Though the battery lasted for only three hours, it was a clear indicator that it was in-fact possible to artificially pace a human being's heart. The man did receive a total of 26 pacemakers, which made us wonder if he was just some experiment for the scientists, but he lived to a ripe age of 86.

So a normal heart has a pacemaker region where the pumping action is synchronised. This pacemaker produces electrical energy (yes the NATURAL pacemaker produces electrical energy) which creates an impulse. This impulse is transferred to the atria (a chamber in which blood enters the heart) which makes them contract and push the blood into the ventricles. After 150 milliseconds, the impulse then in turn moves to the ventricles which in turn



The first external cardiac pacemaker



Embedded systems power medical devices

contract and pump blood away from the heart and as the impulse moves away from each specific chamber of the heart, that section relaxes. Take a moment to understand the complexity of the above process. If medicine is not your thing then still try to understand that the process described is a very delicate one. If the natural pacemaker malfunctions leading to abnormal heartbeats, then blackouts, cardiac arrest, and even death is possible. So what do you do if the pacemaker is faulty? Replace it with an artificial electronic pacemaker. This pacemaker is an embedded device which uses electrical impulses which are delivered by electrodes contacting the heart muscles and so regulates the beating of the heart. A typical pacemaker consists of the computer part and the pacing wire. The computer part has the required electronic components and the battery for the power supply. Pacemakers are now pretty reliable and more than 2 million people in the US alone are living with a pacemaker.

But there is one dark side to having an awesome pacemaker. Recent pacemakers have an added wireless capability which can allow doctors to work on anything faulty in it without having to open it up. Only problem is that if someone has a laptop and has the skills to do it he can hack into that pacemaker and make it issue a lethal shock, effectively killing whoever it is housed in. What a nice way to turn such a beautiful life giving device into an agent of death.

Anti-hacker technologies

Embedded systems are helping create a stronghold of a defense for sensitive security systems. While previously embedded systems were used in the aid of security systems, they would create a single unit out of many processors and hence would centralise the processors and data banks, making them soft targets for hackers. Now what is happening is that even the smallest kernel of information has its own embedded system. That is almost like defending every note in your bank account with an A-grade security system, but not as visual. So in effect, for the hacker to gain access to these sensitive documents, he or she would have to break into each specialised and unique coded security apparatus one painful kernel at a time. You could almost feel sorry for these hackers. Almost.

General data processing

One of the biggest prospects of embedded systems is the processing of data and the elimination of errors. This will be done by integrating multiple processors within complex systems. So as a result, embedded technology can bring together, with maximum efficiency, the functioning of multiple cores in computer systems, hence increasing the speed and accuracy by a great margin.

Here is a quick look at the various industries and how embedded systems contribute to the specific devices:

Consumer function

- Washing Machines to control the water and spin cycles
- Remote controls that accept key touches and send infrared (IR) pulses
- Exercise equipment that measures speed, distance, calories, etc
- Clocks and watches that maintain the time, alarm and display
- Games and toys to entertain people, joystick input, video output

eCommunication

- Answering machines: Plays outgoing message, saves and organises messages
- Telephone system: Interactive switching and information retrieval
- Cellular phones: Key pad and inputs, sound I/O, and communication
- ATM machines: Provides both security and banking convenience

Automotive

- Automatic braking: Optimises stopping on slippery surfaces
- Noise cancellation Improves sound quality by removing background noise
- Theft deterrent devices Keyless entry, alarm systems
- Electronic ignition Controls sparks and fuel injectors
- Power windows and seats: Remembers preferred settings for each driver
- Instrumentation: Collects and provides the driver with necessary information

Military

- Smart weapons: Can recognise friendly targets
- Missile guidance systems: Directs ordinance at the desired target
- Global positioning systems: Determines where you are on the planet

Industrial

- Setback thermostats: Adjusts day/night thresholds, thus saving energy
- Traffic control systems: Senses car positions and controls traffic lights

- Robot systems: Input from sensors, controls the motors
- Bar code scanners: Input/output for inventory control and shipping
- Automatic sprinklers: Used in farming to control the wetness of the soil

Medical

- Apnoea monitors: Detects breathing & alarms if the baby stops breathing
- Cardiac monitors: Measures heart functions
- Renal monitors: Measures kidney functions
- Drug delivery: Administers proper doses
- Cancer treatments: Controls doses of radiation, drugs, or heat
- Prosthetic devices: Increases mobility for the handicapped
- Dialysis machines: Performs functions normally done by the kidney
- Pacemaker: Helps the heart beat regularly

Failures involving embedded Systems

“A failure? So Embedded Technology is not God’s greatest gift to humanity?” Well, slow down young inspired individual. Though they are few and are greatly dwarfed by the positives of this technology, it is always a good idea to know as much as you can about your playing field. While systems failures are generally complex and may not lie entirely in one engineering domain, there have been some instances where the problem has been traced back to a malfunction of the embedded system, be it hardware or a software issue. While these system failures are actually quite rare, it has happened and so deserves to be studied as there is always something to be learned from our mistakes. We have avoided the finger-pointing and biased case studies which you can find all over and managed to pick out some of the actual situations when embedded systems failed to work up-to expectations. Plus this issue is not an advertisement for Embedded Systems so we will show you the slightly weaker side of it as well. Have a look.


Patriot

On February 25, 1991, an American Patriot (Phased Array Tracking Intercept of Target) Missile in Dharan, Saudi Arabia, horribly failed to intercept an incoming Iraqi Scud Missile. This failure caused the death of 28 soldiers and 97 people were injured in the strike.

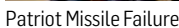
The problem here lay in the embedded system responsible for tracking calculations of the incoming missile. This computer keeps time as an integer value in tenths of a second, and stores it in register of 24 bits length. Later, to

This failure was caused by the 24-bit representation of 0.1 in base 2. So the base 2 representation of 0.1 is a non-terminating 0.00011001100110011001100... The 24-bit register in the Patriot instead stored a terminating 0.00011001100110011001100. Because of this switch from terminating to non-terminating decimal, an error of 0.000000000 00000000000000011001100... binary, or about 0.000000095 decimal was introduced resulting in the accuracy of the time being reduced by 0.0001%. Now this is a very small time difference and is almost insignificant, especially because the tracking of the missile does not depend on the absolute clock-time, but in fact on the time elapsed between two different radar pulses. But there was another issue.

So when the Scud missile arrived, this computer had been active for 100 hours. Remember that we had an error of 0.000000095 decimal from one radar pulse. So to find out the error, $0.000000095 \times 100 \times 60 \times 60$, which is further multiplied by 10 because time was stored in values of one tenth of a second. So effectively, the calculated time elapsed had an error of 0.34 seconds. Since the Scud missile travels at about 1680 meters per second it travels more than half a kilometer in this time. Half a kilometer is enough to be outside the 'range gate'



Patriot Missile Failure



of the Patriot.

So what have we learned from this? Well, the error could have been reduced if the embedded system was capable of handling more than 24 bits, however it would be better to remove rather than just reduce. So the relative time between two radar pulses was being measured. In addition to this the clock should always be reset at the first pulse and stopped at the second pulse. This avoids the accumulation of errors and so also avoids the subtraction of two huge identical numbers.

Pathfinder

NASA launched their space exploration robot, the Mars Pathfinder on December 4, 1996. It was supposed to be a 'faster, better and cheaper' spacecraft, but unfortunately that was not to be. It landed successfully on Mars on July 4, 1997 and then started the collection of data like soil samples, sending images back to earth, and conducting chemical analysis. A few days later though, things had changed. The robot would frequently experience a total system reset because of which it could not function normally and so would always fall to serious data loss.

Pathfinder fell victim to a software system failure called Priority inversion. Pathfinder here was using VxWorks, a real time embedded operating system. This system is responsible to schedule all the tasks and management of all the system resources, for example memory. Using this system, tasks are scheduled in such a manner that one task is never executed continuously, but is instead divided into fragments and interweaved. The tasks are all assigned an appropriate priority. At one time slot the system will perform task 1, and for the next time slot if there is a task of higher priority (say task 2), the system will execute task 2. If there is no task of higher priority than task 1 then task 1 will continue.

For tasks to read and write on, a piece of shared resource called the 'information bus' is used in the same system. What happens here is that data stored by one task into a shared memory, if it has no protection mechanism, will always be overwritten or misread by tasks which are scheduled to perform later. So in simple words, if a task sees that the shared memory is used by another task, it will politely wait for the other task to finish using the shared memory with disregard to the priority levels.

The failure involved three tasks- the meteorological data gathering task (low priority), data transmission task (medium priority), and the information management task (high priority). Before the failure occurred,

the shared memory was in use by the low priority meteorological data gathering task. Again and again before it can release this shared memory it is interrupted because the high priority information management task needed to be executed and it has the next time slot scheduled to it. But, the information management task now needed access to shared memory which is now being held by the low priority meteorological data gathering task. So as mentioned before, the information management waits politely for the shared memory to be released. Now, the meteorological data gathering has to run again. However, before it can finally release the shared memory the medium priority data transmission task wants to join the party and is scheduled. This task then takes extremely long to finish because of all these interruptions. But now the low priority task just cannot release the shared memory because it will not get the chance to be scheduled because of the higher priority tasks which are waiting. Now as a result both the high priority and the low priority tasks are locked and are incapable of doing anything. The system sees that a high priority task is waiting for too long and panics and resets the system, resulting in loss of all that collected data, and numerous groans from NASA on Earth waiting for the results of this expensive expedition.

Ariane 5

Ariane 5, Flight 501, was launched on June 4, 1996, and has the dubious honor of being the first unsuccessful European test flight and of also involving one of the most infamous computer bugs in history. There was a malfunction in the control software, because of which the rocket suddenly went off its assigned flight path 37 seconds after the launch and then subsequently committed hara-kiri due to its automated self-destruct system when the core of the vehicle disintegrated due to high aerodynamic forces. Phew.

An operand error had occurred. This happened due to a very unexpected high value of internal alignment function result call the BH, or the Horizontal Bias. (yes we know that it should be HB but try explaining that to the people who created it). The Horizontal Bias is related to the horizontal velocity which is sensed by the platform, and this value is calculated as an indicator, over time, for alignment accuracy. What happened here was that the value of BH was much higher than what it was expected to be because the initial stage of the trajectory of Ariane 5 was different from that of the previous Ariane, Ariane 4, and this resulted in much higher horizontal velocity values. During execution of a data conversion from 64-bit floating point to

16-bit signed integer an internal software exception occurred. This caused the operand error, and caused the on-board computer to send, instead of real flight data, diagnostic data. This data erroneously commanded sudden nozzle deflections and also induced a high angle of attack

There were quite a few things to take away from this disaster. A clear simulation of the whole trajectory should have recognised immediately the overflow error and should have been performed at the very initial stages like the architectural design or the prototyping stage.

Data overflow software failure where the onboard computer receives only the diagnostic information instead of actual realistic flight data can be understood by viewing what happened to flight 501. There was a backup system of the malfunctioning part in the Ariane , but this was not able to help as it unfortunately used the same software code. Right now there are some notable points which were understood after this disaster

- A test facility should be prepared including as much real time equipment as logistics will allow, injecting realistic data, and most importantly conducting complete and closed-loop testing of the entire system. What also became clear is that although a subsystem can work perfectly on a previous embedded control system, it can still fail as a subsystem in a different new embedded system. In short, the reusability of a subsystem should always be questioned. In the case of flight 501, software for Ariane 5 was considered and trusted mainly because it ran smoothly on the Ariane 4, a clear mistake, as the static code analysis showed the overflow error.
- A sensor must return reliable data in all circumstances. Due to a data overflow of a 16 bit signed integer value the Inertial Reference System (IRS) returned diagnostic data instead of actual flight data. Unreliable data is worse than no data.
- This whole spectacle with the high risks associated with complex systems such as the Ariane 5 and embedded systems fortunately took the interest of politicians, executives, and the general public. This resulted in a large increase in support for research on ensuring the functionality and reliability of safety-critical systems. It is essential that there be a trade off between safety, economic, and environmental aspects as these are all equally responsible and integral to creating a safe, sustainable, reliable system.

HARDWARE ARCHITECTURE

Not all rules are meant to be
broken all at once!

It doesn't matter how great or optimised the algorithms, they're useless without the hardware. You can't make a program written for PowerPC run on an Intel machine or vice versa. While the hardware is also nothing but a piece of junk without the software, it is the hardware which forms a system's base, the bed on which software sleeps. Interestingly, the word 'embedded' is built around the word 'bed'.

The importance of hardware increases manifold when it comes to embedded systems, because the software has to be written in congruence with the hardware. If you diluted the concept of embedded systems to its bare bones, you would have Apple's style of producing devices – binding the hardware close enough to the software so that both of them work in harmony ensuring and the number of problems that would occur stays minimal. Embedded systems, however, have the hardware-software binding at a much lower level, and the hardware-software harmony is not a luxury but one of the core requirements of embedded systems.

To understand the hardware design, let's look at the different parts (call them components, if you will) of embedded systems' hardware. Let's start with the heart and soul – the processor.

Processor

Not just in the embedded systems, in other aspects of computing also the processor is said to be the heart of the entire system – without it, it's guaranteed that nothing will work. Choosing a processor in case of embedded systems is one of the most crucial decisions since it decides almost all other aspects of the system.

Embedded systems are built to handle specific tasks; sometimes, very specific tasks. Due to this dedicated-to-one-task approach, the processor gains all the more importance. You can find a range of processors in embedded systems, which can range from a separate microprocessor to one made as a part of the main board itself.

Separate Processors

You'd be wrong if you thought that the likes of Core i7 or Phenom could be used in an embedded system. But the fact doesn't disqualify other types of processors which can be bought separately as well. In most cases, you



Processors in embedded systems are designed for specific uses

can't buy them directly from a retail shop near your home or on any of the regular e-commerce web sites but they're available separately and can be purchased by shops from a third-party provider. These processors are designed to be used under many conditions and are usually soldered in place into the final product (there are quite a few steps between selecting a processor and fixing it in the final product).

Microcontrollers

Though not always, many a times an embedded system is designed to control machinery.

In such cases, just a processor is not enough. We also need something which can send signals – such as rotate wheels,

toggle switches, control movements of a device and so on – to other non-computing hardware. For such purposes, we don't use microprocessors, but microcontrollers. Microcontrollers are usually processors with hardware control features. They have a set of instruction codes which, when sent, activate some of their pins that are connected to a mechanical arm controller or a motor directly.

Although they bear significant resemblance to each other, microcontrollers and microprocessors are not the same; microcontrollers can do basic computations and can specialise in some aspects but are not as powerful as microprocessors. Typically the processing capabilities of a microcontroller are less than that of a microprocessor. They're majorly used in places where mechanical movements are needed. When there are no physical arms to be controlled, microcontrollers are mostly not needed.



Intel 8051 is one of the most widely known microcontrollers

System on Chip

In most, if not all cases, the capabilities and usage of embedded systems are pre-determined and the kind of workload it will go through is well known ahead of time. In addition, an embedded system needs to be as compact as possible. These two factors motivate another design known as SoC (System on Chip) wherein most controllers and processors (there's more than just one processor, e.g. signal processor) are built into one chip. This reduces the number of chips on the board, saves space and optimises operations for increased speed. A SoC chip designed for a smartphone would contain

a CPU, GPU, Digital Signal Processors (DSP), connectivity controllers, display adapter (different from GPU), GPS processors and a camera input processor on a single chip. This reduces the space these components would otherwise take up and improves overall performance since there are no external controllers needed to control the data flow and conversion. The next time you're in a conversation about how smart phones now are and the many amazing tasks they can perform despite being so slim, think about SoCs.

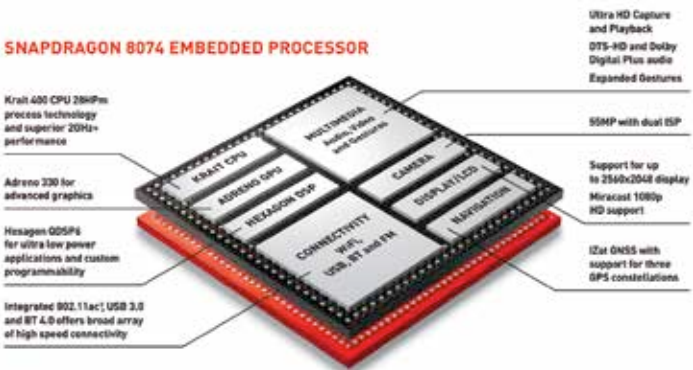
The biggest drawback of a SoC is that since all of the circuitry is contained in a single chip, if say the communications controller were to get damaged, it would require the entire chip to be replaced.

RAM

In most cases, the RAM available on our laptops and desktops wouldn't fit embedded systems for two reasons: firstly, there are no ports to which you could add RAM to, and secondly, since embedded systems need to have everything as compact as possible, the RAMs from desktops/laptops wouldn't fit the criteria. Hence, the RAM too, is built directly on the hardware board. Depending on the need, this RAM needs to be fast and have adequate data storing ability.

Sensors and communication devices

An embedded system can be used to perform either very simple tasks or complex ones. Depending on the variety and number of tasks the system is designed to execute, it would need a corresponding number of sensors. A washing machine panel controller might not need many sensors and can



Snapdragon SoC

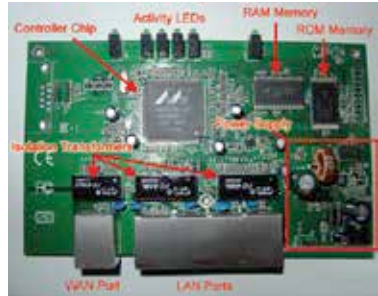
operate with a Timer and a Temperature sensor (to measure the warmth of the water in the tub). On the other hand, a security system or car navigation control system might need many more.

Your phone has a few other (and different) kinds of sensors such as a Proximity sensor, Accelerometer, GPS sensor, Bluetooth, Wi-Fi and so on. All together, they feed data to

the main system (the controller system to which all other components are connected and governed by) which then, based on your current tasks, would take actions. For example, if you're talking on your smartphone, then the input from the proximity sensor will be used to switch off the display while at other times (when you're not talking over the phone), it will remain irrelevant. Similarly, Light sensors are used to dim or brighten your display, while a Wi-Fi adapter can show you the signal quality and available networks.

There are even more types of sensors which can be connected to/used in an embedded system but that would depend on the requirements. For example, a proximity sensor is useless in a microwave oven control system. The requirement of the work environment where the device is to be deployed determines the inclusion of sensors on an embedded system.

However, more the number of sensors more complicated the software and more equipped the hardware must be. This makes you aware of a hardware design constraint of not having enough communication ports through which other devices could communicate. However in the end, the specific design constraints depend on the purpose of the device. Since, we can't talk about all the systems in this tiny booklet; let's go on a walk-through of the common architectural problems and constraints.



Embedded System RAM too is attached to the main board

Architectural considerations

There are a number of requirements which generally pose a challenge to all types of embedded devices.

Component composition

The system architecture should be able to support the composition of a large system using smaller components and dedicated sub-systems. This



Sensors are an important part of an embedded system - shown here is the iPhone4 PCB

is an important consideration for systems which need to support a change in any of those sub-components. Consider the example of a microwave oven – if the temperature sensor starts working erratically, then one might need to change just the temperature sensor instead of changing the entire control panel. In such cases, the number of components and the option of upgradability must be thought out well ahead of production.

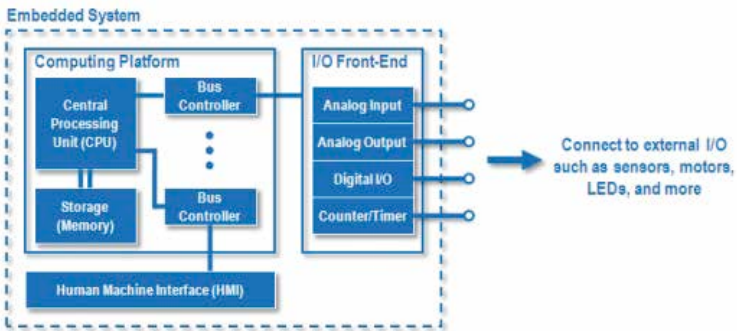
Components in any system (even non-embedded systems) serve as a window of interaction between the system and the outer environment. Taking the microwave oven example again, the controller wouldn't be able to know the temperature inside the heating cabinet if it were not for the temperature sensors. Components are connected to the main system via interfaces. Depending on the need, the interface can be either technology dependent or technology independent. A technology dependent interface connects the component to the board; this interface is not widely used and serves the purpose of controlling, programming and debugging the component alone. A technology independent interface can control the overall functionality of the component, and can send and receive data.

Networking and communication

While embedded systems which are used as networking devices must ensure networking between interfaces, other programmable electronic systems which use multiple components often have to do that as well. An embedded system must ensure that different components which are connected to the system are able to communicate with each other. One of the best examples in our day-to-day lives of this feature is Wi-Fi hotspotting available in smartphones: the phones can receive and send data to the network service

provider using their 2G/3G/4G communications radio component and can receive and send data to the Wi-Fi adapter which then can be used by the Wi-Fi adapter of another device (say, a laptop) for connecting to the internet. Both of these adapters can communicate with each other internally.

At the same time, it is necessary that the computation and communication parts of the networking scene be kept separate. The data brought in by the communication systems shouldn't be completely decoded by the communication component itself. Instead, the data is computed upon by the related processor of the embedded system. A design like this ensures that the communication gear is kept intact while the core system is changeable or vice versa. For a designer, it helps with debugging and finding problems



There can be many ways to get it done but inter-component communication is an integral part of embedded systems design

by isolating the two parts. Overall, it reduces the design complexity. Realistically, this ensures that feature updates can be sent to the software without affecting the hardware.

Security

This aspect of embedded systems, if neglected, can prove to be costly and dangerous. Most embedded systems of today are capable of establishing network connections to other systems or to the internet. Any system which transmits data should be able to communicate in both, protected and unprotected modes. This places a requirement on the networking and communication system to be able to support secure protocols. For example, a Wi-Fi transceiver should be able to communicate over open networks as well as over secured connections. However that's just one side of security.

One of the security constraints of an embedded system in comparison to other normal software systems is its difficulty in procuring software updates which can fix known issues (This is called ‘intrusion containment’). Let’s elaborate a bit on this with a comparative example: Windows XP and below had a list of severe vulnerabilities exposed to the general public via the (in)famous Internet Explorer. This vulnerability was caused by two things – usage of ActiveX controls which allowed an unusual amount of control on the system by a remote attacker, and secondly IE’s deep penetration into the Windows OS. However, these issues associated with IE were resolved with patches and service packs which ensured that the users’ computers were protected against further threats. In due course of time, IE 7 was released as separate software for Windows XP and it filled the gaps left by IE 6 to quite an extent.

The two security issues here that stand out in terms of importance to embedded systems are:

1 Intrusion Containment: When an intrusion happens, the hack shouldn’t be able to corrupt systems other than what was compromised. In case of Internet Explorer, its depth of connectedness to Windows opened new loopholes and allowed attackers and malicious websites to read and write data or run malicious code on the system. If only IE were to be just another application installed on top of the OS with no deeper level integrations, the threats would have been smaller than they were. In case of embedded systems, it’s vital that the components are isolated from other components from the attack perspective and a compromise of one component (let’s say the communication component) shouldn’t affect other components (e.g. motor controls).



Flashing a phone for installing updates can be complicated even though they are not the most complicated or critical embedded systems

2 Ease of Update: It’s one thing to install an update for Windows and quite another to install one for an embedded system. The difference in level of ease (or difficulty) can be understood by comparing a Windows installation

on your PC against a ROM installation on your Android phone. Without a doubt, installing Windows is much friendlier than flashing ROMs. Since embedded systems are not normal PCs and the software written for them has to be correct and corruption free to the last bit, installing an update for them is a lot harder.

These security issues must be kept in mind right at the outset when designing the hardware. While software level updates can be made easier, there must be a limit to the frequency of updates due to various reasons such as high difficulty level and risk, legacy code, need to be always-up and stringent performance requirement. At the same time, the software layer of the embedded system must be built only after the hardware part is completed. A hardware change can bring in significant problems for software writers and can delay production or cause errors to creep in. This is another reason why components have to be carefully selected and organised in an attack-isolated fashion.

Availability of common time to all components

There are two possible interpretations of the phrase ‘common time’:

- 1 Common CPU Time: All components attached to an embedded system need to use the same CPU time. Assuming you’re watching a video on your smartphone, your data connection and video decoder would be both putting some computation load on the processor. Also, other components such as Bluetooth and GPS antenna might be sending interrupts to the processor at the same time. Since you can’t simply change a processor in an embedded system, the mechanisms to allow interrupts and (if needed) limit resource usage by various components need to be thought out when designing the system. This requirement is also referred to as ‘CPU Time Sharing’.
- 2 Common Timestamp: If there’s a need for one component of the system to interpret messages sent by another component periodically, it becomes necessary for both the components to read time from the same clock and calculate time differences. In case they use different clocks, the receiving component will reject the messages from the sending component if the sender’s message has a timestamp which is from the future (i.e. the receiver component’s clock is running behind the sender component’s clock).

Interestingly, the common timestamp feature can aid in system security by saving from ‘replay attacks’ wherein the same message is sent more than once and can cause unwanted results on the receiving end.

State awareness

Imagine your Windows installation went bonkers and started misbehaving. While reinstalling the OS is an option, it's the last one. You'll perform a number of checks such as running a virus scan and cleaning your registry; if all else fails, you're likely to use System Restore feature built into Windows. A real life-saver, System Restore will restore Windows to the state it was in at an earlier date and time. Yes, you can lose some programs and maybe some settings, but you won't have to take the pain of reinstalling the OS, and then reinstalling all the software on top of it.

What System Restore is basically doing is termed as 'state awareness' i.e. the process of capturing the state of the system including the installed programs and drivers, Windows settings and other crucial pieces of data. Later should you face a problem, you can choose one of your saved states that you know was working fine and system restore will automatically bring your Windows installation back to its saved state. Cool, eh? Not so much when you realise how much of a difficult feat this is to accomplish with embedded systems. Remember that installing anything on an embedded system is a lot more difficult than on a PC.

In case of an embedded system, component or system failures can cause huge losses, sometimes of lives. It's important for such systems to be able to go back to their original state (or a pre-saved state) if a transient failure occurs. Transient or impermanent failures can happen due to a software layer bug or voltage fluctuation. Such failures are impermanent and don't cause hardware damage but can temporarily disable the system.

In such cases it's important for the system to be able to recover itself to a previous workable state or its original state. This can be achieved by transferring the state to another system periodically or saving it on a storage unit locally. However for a state-aware design, it's important to save the state of components as well (along with the main system). As you might guess, it's the job of the software layer of the system to detect a corrupted state and be able to restore one from the past. While much of the state-aware design needs to be implemented at the software level, it's important that the hardware of which we need to save the state of has hardware features to support this.

Robustness

Robustness should not be confused with 'recoverability' which is the capability of recovering from a failure (and state-aware design does take care of this for the most part). Robustness is the attribute which helps prevent failure

in a system. While much work for this feature must be done in software, there are hardware features which need to be in place for a more robust design.

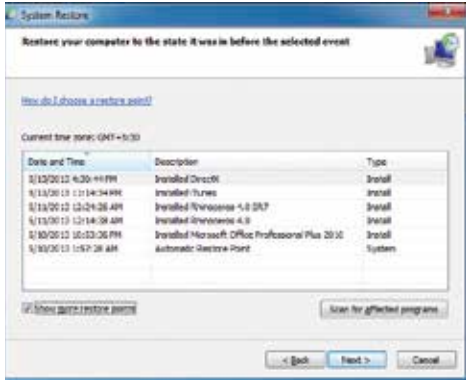
Robustness, at the hardware level, requires techniques to ‘isolate faults’. Since all components on the system consume the processing power of the same CPU, it can be difficult (and less effective) to design a system whose fault detection system is located centrally and takes care of all components. On the other hand, if each component comes with its own fault detection system, it simplifies the design at many levels and offloads ‘fault detection’ from the main processing system thus improving both, performance and robustness.

There are two concepts you need to understand for better fault isolation:

1 Temporal Partitioning: It means that one component cannot affect the guaranteed availability of communication resources available to another component. Here, by communication resources, we mean resources such as time of availability, latency, noise etc. Imagine your phone stops responding while trying to connect to a Wi-Fi hotspot because the Wi-Fi adapter clogged resources available to other parts of your phone. Temporal partitioning avoids this.

2 Spatial Partitioning: This not only improves robustness but also improves security. Spatial partition design ensures that the data integrity is maintained during communication between two components and that a third component cannot manipulate (or in some cases, even read) that data. This decoupled design ensures that if for some reason, one component is compromised, it won’t allow data to be stolen from other data streams. In other words, if your phone’s Bluetooth were to be hacked, the data you’re transferring via 3G network would not be available to the hacker/attacker.

Once again, both these features need a plethora of support from the software but the hardware design must support them as much as possible. Although, we’ve cited the example of devices used in our day-to-day activities to keep it as simple to understand as possible, there are many other embedded systems where these design features are highly anticipated.



Windows System Restore saves the states of the OS at different times

Fault Diagnosis is yet another feature which an embedded system should incorporate lest it's a simple home appliance where a fault wouldn't cause heavy losses of life or property. Fault diagnosis is usually done on the core system (not on components) but can be done otherwise as well. It requires the system to build assertions and test them. If they fail, then a fault recovery sub-system can be activated. Let's take the example of a home management and security system: if the temperature sensors and infrared cameras don't detect any rise in temperature and the electricity regulation system doesn't report any problem but the smoke detector reports smoke being generated then it might fail (depending on the design) the assertion test for 'fire outbreak' and call for fault diagnosis for the smoke detection system.

Resource management

Resource Management is mostly a software level task. However, as we said at the beginning of this chapter – software is impotent without hardware support. The hardware must support features which enable better



Fault Isolation is important to ensure robustness of an embedded system

software to be built. If the hardware doesn't send enough information or sends it in a difficult-to-process format to software, the design won't be optimum. The hardware used in the system should be able to work over 'messages' and be able to send data to the main system in a similar format. Since polling is not a good way to utilise computing resources, it should be able to send interrupts and the main system should be able to handle them. Processors should be able to execute multiple jobs in a round-robin style by dividing time between jobs (multi-tasking) and be able to call the supervising program (the OS) at times. These hardware features open possibilities for higher quality software for the system.

CHALLENGES IN HARDWARE DESIGN FOR EMBEDDED SYSTEMS

Designing hardware for embedded systems is challenging, because the designers have to be very, very careful, for a lot of reasons. Think of a system such as your Wi-Fi router, which is just supposed to do one job well (i.e. routing). Here, ensuring the quality and stability of the design is essential. If your

Wi-Fi router is up and running as soon as you switch it on, it's because someone probably worked very hard to ensure that it behaves flawlessly.

Designing hardware for embedded systems requires a deep understanding of the real-world scenarios within which electronic systems are used. If you're planning to embark on a full-fledged career in this field (more on this in a later chapter), this chapter will help you understand the issues you'll be up against. Even if you're not an electronics expert, we aim to help you understand how critical systems are designed. Let's start off with the heart and soul of any system – the processor.

Design challenges when making a processor

Power consumption – Why more is not always better

Like we already said, you're not going to find a Core i7 or Phenom in an embedded system. Why? Because typically embedded systems have processors that are very low power, which also means low complexity. A PC has enough power to feed these x86 monsters, but embedded systems don't.

An x86 (or x64) processor belongs to a kind of processor family known as CISC which stands for "Complex Instruction Set Computer". This type of processor has a more complicated instruction set as compared to another family of processors that's called RISC (Reduced Instruction Set Computer). It's RISC processors that are mainly used in embedded systems. A CISC processor is easier to control and command because it carries out some of the more complicated functions automatically, while a RISC processor isn't as smart. The programmer or compiler is tasked with the job of writing instructions for the processor to help it carry out more complicated jobs.

Again, you might wonder why bother with RISC processors, why not just use CISC processors all the time? This is because simpler RISC-based systems cannot afford to use as much power as a CISC system. How would you like it if your simple router used as much power as a PC? Besides, if we used CISC processors everywhere, everything would cost that much more and be about as big as a PC. This is why ARM processors are majorly used in mobile phones – ARM processors are built using the RISC instruction set architecture.

Word length – For how long can it think?

Let's assume an 'elevator control' system is our target system. How much data would the system need to process in single cycles? The largest number the system would have to deal with would be the highest number of floors the elevator has to travel up. Assuming that the number is 50 (for a 50-storey building), you'd

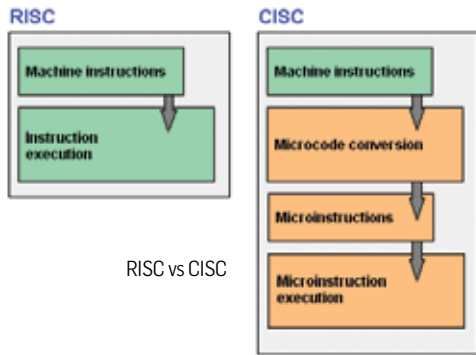
need to store all the numbers from 0 to 50 and the processor should be able to process those numbers. In such cases, an 8-bit processor (one which can process 8 bits in one clock cycle) is more than enough since it will be able to process any number between 0 and 127 (128 numbers) without having to resort to any extra tricks. Given the fact that elevator control systems only perform two types of operations – comparison and check, this processor should work fine enough.

This however may not be true for a Wi-Fi router; the reason being what it does – routing. One essential function of data routing is ensuring that the data packet received is intact. Now, most systems work with IPv4 which comes with a 16-bit header checksum

to ensure data integrity.

If the processor in this case were to be 8-bit, calculating and comparing the checksum would need extra computing cycles along with some programming tricks to accomplish the task, and that would consume more power. A 16-bit pro-

cessor would be better in this case. Hence, it's important that the processor in an embedded system match the word-length requirement to get the job done without needing more computing cycles, electric power or tricky programming.



Memory addressability

In most cases, the word length of a processor is the same as the number of bits that it can address on the memory bus. While this is true in most cases, there are always exceptions. One of the best examples is the PAE extension on most desktop Intel processors which can allow the processor to work with more than 4GB of memory (addressable by 32 bits) while keeping the word length as only 32 bits. With that small example, we mean to convey that the same can be the case in an embedded system as well.

An 8-bit processor might have to work with more than 256 bytes of data and hence memory addressability is another important factor. If the selected processor can't address the amount of data the program has to work upon, then the system can crash, hang, or do something that can produce catastrophic results.

Number of registers

Did you know that a graphics card can have as many as 3072 processors and each processor can have up to 255 registers? While the number of processors is probably enough to amaze you, what is more amazing is the number of registers available to the processors! If you compare this number to what's in a normal PC processor, it's huge. Most x86 processors have only about 20 registers to work with. If you've ever worked with assembly, you'd appreciate the luxury of being able to work with 255 registers at one time. If you haven't, take our word for it, it feels spacious. Very spacious.

For those of you not familiar with the term 'register': A register is a very small and extremely high-speed memory that the processor can directly access, and requires exactly one cycle for the processor to retrieve data from it. Thus, obviously, the more registers your program uses, the faster it runs. Also, a program using the registers will run faster than one using the system RAM. The truth is, RAM can be hundreds of times slower than accessing data from registers.

Let's recap and take a look at the list of challenges one could face when designing or selecting a processor for an embedded system:

- 1 Power Consumption
- 2 Processor type – RISC / CISC
- 3 Word Length
- 4 Memory Addressability
- 5 Register count

And this is just for the processor!

Depending on the specific use-case, the right mix of the above can help get the best results. Again, the processor alone is like a severed head; other components are equally vital – each of them with their own design challenges.

Design challenges when building RAM

Without relying on your memory, you'd get very little done. The same applies to the electronics world. Every piece of equipment that can calculate or give commands should be able to store some data in the system's primary memory, also known as RAM. You probably already know that RAM is a fast but volatile memory closer to the processor, and is much faster than secondary storage in terms of the time taken to access it.

In most (if not all) cases, embedded systems have the main memory embedded onto the main board itself. The performance requirements are strict and real-time and there's no scope for altering or replacing the com-

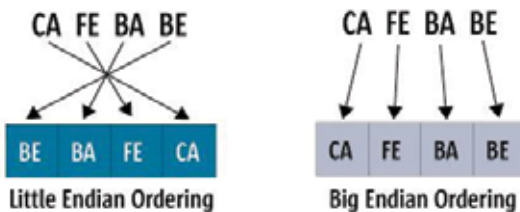
ponents once they're set into place. The performance requirements in most cases are RAM that matches the processing speed, so that the data is made available as soon as the processor needs it, thus avoiding delays in processing.

Now, you can't just use a RAM stick from your desktop or laptop in an embedded system. Though theoretically the same, there are additional factors that make this impossible:

- 1 Normal desktop/laptop RAM modules are sold separately and can't be soldered into an embedded system's main board directly.
- 2 Power consumption is always an important issue. There are lower power RAM modules available and they're easily recognisable by the 'L' suffix such as DDR3L. Again, depending on the specific requirements, the RAM might have to be designed for very low voltages or those voltage levels which normal RAM modules are not built for.
- 3 The method of accessing the data from the memory is equally vital. For example, if the processor accesses data in a 'Big-Endian' style, but the data is stored in RAM in 'Little-Endian' style (Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory.), it will cause some amount of processing to kick in to ensure the correct orientation of data. Hence memory controller and RAM must always be selected/designed with this requirement in mind.
- 4 If you've seen RAM being sold online, you may have come across the term 'ECC'. It stands for Error Correction Code, which is usually a missing feature on normal RAM modules you can buy from brick-and-mortar stores. The ECC feature ensures that the data stored in the RAM modules is always correct. In situations where the accuracy of data matters a whole lot more than normal, ECC memory is normally required.

We've already mentioned that a graphics card utilises a whole lot of processors and registers, but did you know that one of the cleverest and most important parts of a graphics card design is its RAM? A normal desktop or laptop RAM

serves data to the processor in the sequence of requests, which is why companies came up with the idea of multi-channel RAM modules. On the other hand, a graphics card utilises the very clever



Little Endian vs Big Endian memory layout

technique of serving all the processors with their requested data within one cycle. Hence the processors can go on computing huge amounts of data without having to wait their turn to get more data.

In addition, graphics cards or GPUs come with 'texture memory' which is memory specifically set aside to ensure faster read-write operations and faster computations for 3D renderings. Since the GPU is designed to process 3D calculations fast and efficiently, it will have local access to the texture images, allowing it to complete a scene much faster than having to use memory stored in another physical location within the computer. Such an implementation would obviously be very beneficial in critical conditions where time is of utmost importance.

Just imagine a failed Curiosity Mars rover landing due to slow radar readings and estimations. Had that happened, it would have set the U.S. back by more than \$2 billion and cost a delay of more than a couple of years to investigate the Red Planet. RAM speed in embedded systems in the context of an interplanetary project is especially very important. Every single detail can make a huge impact in such situations where milliseconds matter and there's no room for error.

Displays – Challenges and considerations

In many cases, an embedded system doesn't need displays and so displays have no relevance in such products. In products that do require visual indicators, a small panel of LED lights usually suffice - e.g. the audio system in your car or the control panel on a microwave oven. There are other applications of embedded systems for which CRT-type displays (medical equipment and electronic frequency readers) are used.

The type of display required will determine whether you'll face difficulties in designing this component. For systems with minimal visual indicators, the circuit required to control the display is controlled by a chip, and this can also be done directly by the processor.

However, certain systems may need extensive use of displays. The best example is your mobile phone. Yes, this would include every phone from the most basic to the likes of BlackBerrys and iPhones. If you care about the details of a smartphone's specs, you might have observed that most Android phones (since they run on a wider variety of hardware than other mobile OSes) are built around a specification predefined by a SoC (System-on-Chip) design company which requires details about the processing power, memory included, memory handling capabilities, communication ports available, on-board graphics processors and maximum supported resolution of the display to be mentioned. While a

SoC is not just a display, but a complete embedded system on its own, an example specification would help you get the idea. Here is a sample of such a specification: <http://bit.ly/18QF8jf> The page also contains links to other chipsets which have been used in different smartphones. Go through them to understand what goes into the making of a smartphone (but that's another story).

An embedded system can't be too flexible in terms of display resolution and frame rates, for both of which the designers must put limits before actual assembling or production takes place. Taking the example of MediaTEK MT6575 (for which we've provided the link earlier), you can't attach a 1080p screen to the chip; mainly because the processor, system bus and memory aren't built for that.

A display or screen needs signals for every frame to be fed to it. The signals are fed by the display adapter which is attached to the system. The display adapter has two requirements of its own from the system – supply of display data, which is to be converted into signals for the display and power consumption for signal processing. Display data is usually kept in a part of the main system memory (this is analogous to a PC with an on-board GPU) or in a dedicated graphics memory (and this is analogous to having a discreet graphics card in a PC). The bigger the display is, the more power and memory it requires.

When it comes to graphics controllers, you have two options:

- 1 Basic graphics controller: The graphics memory is shared with the main memory of the system. This option is good for lower resolutions with less power hungry requirements.
- 2 Dedicated graphics controller (such as an NVIDIA Tegra chip): The graphics memory is separated as is the computation. This is good for bigger displays which have more power requirements.

Communications and networking

Home appliances are one of the most widespread uses of embedded systems. Many of them don't need any networking features, but all that's about to change as we progress towards a more connected home. While you wait for that to happen, let's take a look at the various applications of embedded systems in devices facilitating communications, and the challenges associated with them.

Mobile phones

Most mobile phones have the communication systems connected on the main board. While integrating them is a challenge of its own, there's no major effort needed these days, as SoC designs that do this are already flourishing.

Networking equipment

Almost all networking equipment is made up of embedded systems, and the primary reason for this is the need to be as fast and reliable as possible. One of the biggest challenges here is the ability to compute the path (for routing). Most of the data-link layer responsibilities are taken care of by the individual port adapters and they don't put any significant load on the system. However, the route calculation can be done on dedicated hardware which speeds up the packet delivery by orders of magnitude. Small Wi-Fi routers in home set-ups use embedded chips for the work. The challenge with dedicated hardware is upgradability and creation of routing logic on hardware. This is more so because there are multiple routing protocols involved and each have their own way of getting the job done.

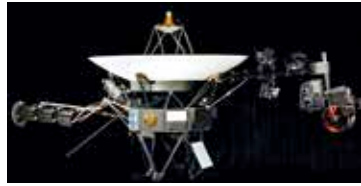
When complex topologies are to be handled (on higher-end routers), it becomes a challenge to design a chip which can get the job done. Also, the router has to be programmable and the system as a whole needs to run an OS which is familiar to the network administrator (in most cases, this ends up being Linux). This complicates the design because the system would then have to incorporate a more generic processor which can run the OS. The closer the routing logic to hardware, the better and faster the router works.

Inter-planetary systems

Voyager 1 is the first man-made object to leave the solar system and enter inter-stellar space. It has been sending information to NASA scientists about the solar system for the past 35 years! Mars rovers are more examples of ultra-rare systems that are working in an entirely different planet, but controlled from Earth. Satellites of all types are filling our lives with more information and ease than ever before. These systems are short on power. While you might get sunburnt on Earth, you would freeze to death in the low temperatures beyond the asteroid belt. The fact is that the average maximum temperature of Mars is about 27 degrees Celsius, and minimum temperatures go down to less than negative 100 degrees Celsius! Satellites have to deal with a much higher variation in temperatures, since they face much higher maximum temperatures when the sun shines on them.

Most day-to-day communication devices have operational temperature limits because extreme temperatures can damage the equipment. At the same time, power requirements of interplanetary systems are higher than any other wireless device due to sheer distance they have to communicate across (measured in thousands of miles). There are basically two requirements for an interplanetary or extra-planetary system:

- 1 Resistance to temperature and temperature changes to ensure that the equipment doesn't fail.
- 2 Enough power to transmit signals to Earth or the Deep Space Network of satellites.



Voyager 1 is one of the most important space probes ever built

Challenges in Upgrades

There are hardly any upgrade challenges at a hardware level because many a times, devices are replaced instead of upgraded. Yet, a system which is controlled by an embedded device can receive an external hardware upgrade, e.g. the camera of a phone can be replaced by service centre personnel. In situations where upgrades or hardware changes are required, the design must be highly modular and allow for the replacement of parts that are not embedded on the main board.

And then there are software upgrades. Think of your iPhone being upgraded from iOS5 to iOS6. Though this chapter covers hardware challenges, software goes hand-in-hand with hardware as far as enabling updates is concerned. For example, if the phone is to be upgraded to the next version but not beyond, the hardware included need not be very powerful, but just powerful enough to work for the anticipated improvements in the newer version of the software. The case with networking equipment is largely the same.

Heat Dissipation

No matter what methods you use, as long as electronic equipment runs, it generates heat. While you can use a monster cooling fan for a gaming PC at home, embedded systems have limitations. Embedded systems can range from low power devices running on batteries to those powered by constant power supply from the wall socket.

Heat dissipation is a major issue for any system; if not handled properly, excess computation can melt down the system just by the heat it generates by itself. In certain cases, where there is enough space, fans can be used. However, equipment designed to be small, silent or powered by a battery can't use the technique and must find other ways to stay cool. Punching holes at the top of the body to let hot air escape is one of the techniques commonly used. Systems that need to be as compact as possible (such as a mobile phone) must use better heat conducting materials and design, and ideally radiate as much heat as possible through the outer casing.

SOFTWARE SYSTEMS

This chapter will cover RTOS in detail explaining various internal modules of the system – kernel, scheduling, IPC, memory management, I/O and time services. This will be followed by a discussion on various embedded operating systems currently available in the market and an overview of software developed and deployed in embedded systems.

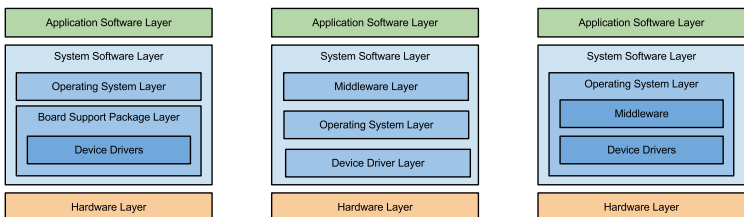
Embedded software

Software running on embedded systems is vastly different from those that we run every day on our desktops and/or laptops. Since software in most cases is customised for the hardware on which it is to be deployed, software for embedded systems must be designed keeping in mind the limitation and capabilities of the platform. Embedded systems impose constraints of time and resources. Due to the timing constraints, embedded systems are often referred to as real-time systems i.e. the correctness of a computation not only depends on the logical correctness of the result but also the time at which the result is produced. The system is said to have failed if it doesn't meet the timing constraints. Resources on such systems can vary from 6KB ROM and 4KB RAM with a CPU operating at 8MHz to specifications that can rival modern computers but which are limited by the power available.

The eponymous operating systems that power these systems are designed to be compact, reliable and efficient at utilising the resources. Such OSES are not required in the case of small systems which can be easily controlled by a simple set of instructions stored in the device's ROM. However, they're a requirement for medium to large scale systems which must perform multiple tasks in real time. Often these operating systems are referred to as real-time operating systems (RTOS).

Embedded / real-time operating systems

For an embedded system, the OS is a set of libraries which sits either over the hardware, over the device driver layer or the BSP (Board Support Package) layer. It acts as an abstraction layer between the hardware and the software running on top of the OS so that it doesn't have to deal with the hardware's idiosyncrasies. This also allows developers to come up with better software and middleware. It goes without saying that it has to do all these operations keeping the constraints associated with real-time systems in mind – it has to ensure timely completion of tasks. manage



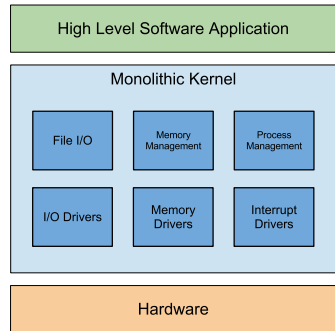
Embedded OS Models

various hardware and software resources and ensure safe and reliable functioning of the system.

Embedded OSes come in three flavours: monolithic, layered and micro-kernel. These models differ based on the internals of the OS's kernel along with the additional system software incorporated in the OS.

Monolithic

In case of a monolithic OS, the middleware and the device drivers are included in the kernel itself. In most cases this OS is usually a single executable file which contains the above mentioned components. Monolithic OSes are in most cases more difficult to scale, maintain and debug than their architectural counterparts.



Monolithic OS

Usually, a more popular alternative to this type of OS is used, it is called the modularised monolithic model. In this case instead of one huge file, the OS is broken down into separate files called modules. Each module is responsible for one of the OS's many functionalities. In this case only the modules needed are loaded into the memory and it also makes scaling, maintenance, modification and debugging easy.

Linux is a popular modularised monolithic non-embedded OS. Jbed RTOS, μ C/OS-II and PDOS are examples of embedded monolithic OSes.

Layered

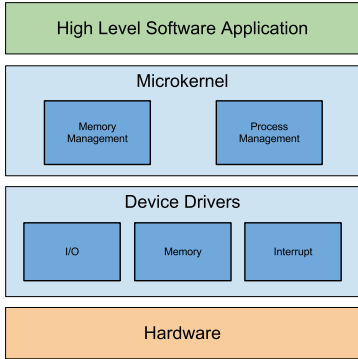
In a layered system, the OS is divided into hierarchical layers, where the upper layers are dependent on functionality provided by the lower layer. Like a monolithic OS, this system is made available as a single, large executable file which includes the device driver and the middleware along with the kernel. But due to its layered nature it's easier to manage and debug such systems. One of the drawback of this design is the overhead incurred due to the overhead added by the API provided at each layer.

DOS-C(FreeDOS), DOS/eRTOS, and VRTX are all examples of a layered OS.

Microkernel

This is the most popular OS model among the off-the-shelf embedded OSes. Often called the client-server OS – the main OS is stripped down to the min-

imal functionality usually limited to schedulers and memory management subunits. The remaining functionality including the middleware and device drivers are completely abstracted out of the microkernel. Microkernel based systems are infinitely more scalable and manageable since the required components can be added in dynamically. It's also much more secure as all the non-OS specific code has been moved to the client, and there is a precise demarcation



Microkernel OS

between the client and server memory. The drawback with this architecture is the overhead associated with the communication between the micro kernel module and the other modules moved out of the kernel. Context switching is also costly affair in this case as compared to the other architectural paradigms.

Examples of microkernel OSes are OS-9, C Executive, vxWorks, CMX-RTX, Nucleus Plus, and QNX.

Kernel

Like any other OS, the heart of an embedded OS is its kernel. The kernel is the nerve center of the operating system and is responsible for:

- 1 Booting
- 2 Task Scheduling
- 3 Memory Management
- 4 I/O System Management
- 5 Standard Function Libraries

Embedded OSes will differ in which components they include from the models shown above, but lowest common denominator is the kernel, all the OSes have a kernel at the very least. We'll discuss all the kernel components in the following sections. Before that a heads up about the standard function libraries, usually such systems have very limited space so there is no question of a full featured library likes of libc existing in the memory. The prerequisite of a function being included in the library is that they're small and really important.

In most of embedded systems, the bootloader will prep the kernel and load it into the memory. Kernel is responsible for initialising the system and ports, setting the global data items, initiating hardware timers and acti-

vating the scheduler. The kernel is then swapped out of the memory in lieu of the scheduler, except for some important library functions, and will start executing the child tasks.

Scheduling

The smallest unit of execution in an RTOS is called a task. A real time task is generated in response to an event. This event can be an internal event like a clock interrupt generated by a hardware clock every few milliseconds or an external interrupt generated when a predetermined event occurs like a sensor value going above or below a threshold. Nearly all real time systems consists of multiple real time tasks. The time constraints on different tasks will be different. It is onus on the scheduler and the operating system as a whole to make sure that the tasks are executed on time.

In this section we'll outline some fundamental task scheduling algorithms. It'll not be possible to discuss the nitty-gritties of each algorithm due the limitation of space but we'll list some resources at the end of this section, for those of you who want to dig in deeper.

The schedulers are broadly categorised into 2 main categories – clock driven and event driven. We'll discuss some of the important members of these broad classifying categories.

Clock driven

Clock driven schedulers are those for which the scheduling points occur only at timer interrupts. They're also known as off-line schedulers as the schedule is fixed before the system starts running, in other words the scheduler pre-determines which task will run next. As a result such schedulers incur very negligible runtime overhead with regards to making a decision for which task to schedule next. But due to their intransient nature such schedulers are ill-equipped to handle dynamically generated tasks. We'll briefly discuss the two important clock driven schedulers

Table driven

Table driven schedulers usually pre-determine which task to run next and store the result in a table at the time the system is designed or configured. The schedule is not automatically generated by the scheduler rather the application programmer is completely in control of the schedule and is free to decide the order in which the tasks will be executed and store it in the schedule table.

The timer has to be reset every time a task starts to run. A typical real time tasks lasts for a few milliseconds. This leads to significant overheads as the call to reset a timer takes place every few milliseconds, which in turn leads to degraded performance. A more prevalent and better clock driven scheduling algorithm exists which we'll discuss next.

Cyclic scheduler

A large majority of small scale systems in the industry rely on cyclic schedulers. It's easy to program, maintain and is much more efficient as compared to table driven schedulers. A cyclic scheduler repeats a pre-computed schedule. The schedule constitutes of major cycles and which in turn contain minor cycles. The schedule needs to be stored only for one major cycle, as all tasks repeat themselves in subsequent major cycles. A major cycle is divided into multiple minor cycles which are also known as frames. The scheduling point for a cyclic scheduler is the frame boundary i.e. a task can only be executed starting from a frame boundary. As this is a clock driven scheduler, the frame boundaries are defined by clock interrupts. Each task is assigned to run in one or more frames. This assignment of a task to a frame is stored in a schedule table.

Interrupt driven

A fundamental drawback with clock driven schedulers is that firstly it becomes very complicated to generate a schedule for a large number of tasks and it's very difficult to arrive on an optimal frame size. This is where interrupt driven schedulers shine, they're much better at handling sporadic and aperiodic tasks. In an interrupt driven scheduler the scheduling points are defined by task arrival and task completion events. These schedulers are normally preemptive, which means that they stop a low priority task when a high priority task arrives. We'll discuss 3 important interrupt driven schedulers.

Simple priority based scheduling

Also known as foreground-background scheduler, it is the simplest priority based preemptive scheduler. There are only two priorities at which the tasks can run, the real time tasks run as foreground tasks or at high priority. The dynamically generated sporadic or aperiodic tasks are run as background tasks or at low priority. At a scheduling point, the foreground task is scheduled which is at a higher priority, the background task can only run if no foreground tasks are ready.

EDF (Earliest Deadline First) scheduling

This algorithm is very intuitive and extremely easy to understand. At each scheduling point the task with the closest deadline is executed. EDF has proved itself to be the optimal single-processor scheduling algorithm for real-time systems. If a set of tasks is not schedulable by EDF, then no other scheduling algorithm will be able to feasibly schedule a task set. EDF is generally implemented using either heaps or FIFO priority queues.

RMA (Rate Monotonic Algorithm) scheduling

RMA is a static priority algorithms, by a static priority algorithm we mean that each task is given a static priority based, in this case based on it's rate or period as opposed to other cases when the priority based on deadline, which can vary. Lower the occurrence rate of a task, lower is the priority assigned to it. A task having high occurrence rate accordingly given a high priority. RMA is the most optimal static real time task scheduling algorithm. RMA is the most common real time scheduling algorithm. Nearly all the commercially available operating systems support RMA. Tasks are divided among multiple queues based on their priority levels. All tasks at a single level are then scheduled using time slicing or in a round robin manner.

Inter process communication and synchronisation

Different tasks running on an embedded OS share the same hardware and software and it's often the case that they're dependent on other tasks to provide some of the functionality for the work that they have to undertake. For these reasons embedded OSes provide various methods and mechanisms for communication and synchronisation. These mechanism are based in some combination of shared memory, message passing and signaling.

As mentioned earlier most of the advanced scheduling algorithms use preemption, i.e. a higher priority task swaps out a lower priority one at a scheduling point. This opens a new can of worms, that stem from the fact that control is taken away from a task which is not ready to give up control. For instance, if a task is writing to a shared memory location and was preempted, the next process would see corrupted data when it tried to access the same location. Or, if a task where waiting for input and was preempted, the actual signal for the task would be lost. The solution to these problems is to use constructs like mutexes and spin locks in the critical regions of the code to prevent preemption when a task is undertaking some critical operation.

Mutex

Mutex is a portmanteau for 'Mutual Exclusion'. Mutex are commonly found in all kinds of operating systems and one of the methods for synchronising two tasks. Mutexes follow several rules.

- 1 Mutexes are system globals
- 2 They can be owned by only one process at a time
- 3 If a mutex is already all allocated, then the function which requests the mutex for a task and by proxy the task will block until the mutex is available

Spin locks

Spin locks are the quick synchronisation mechanisms. As the name suggest a task spins in a loop until the lock opens or the condition for the lock is false. This lock is provided by the CPU so it is only possible to use it when the CPU supports the test and set idiom or gives exclusive access to a thread or a task by either masking interrupts or locking the bus.

Memory management

In their most basic form all the tasks running on a compiler will be set of instructions to the processor, now the processor will only execute the code that resides either in it's cache or it's RAM. Multiple tasks share the same memory, even the OS resides in the same memory. So the operating system kernel must come up with a way where a task's code remains independent of other tasks and also manage it's own code and make sure that no task code can manage this. These are the functions that are the responsibilities of the memory management module of the operating system.

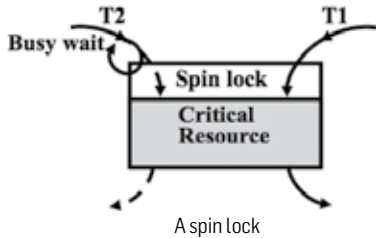
In most systems, physical memory is composed of two-dimensional arrays made up of cells each addressed by a row and column number. Each cell stores 1 bit. On the other hand the OS treats the memory as a one-dimensional array known as the memory map. The conversion between the logical and the physical address is carried out by the MMU on board or the operating systems.

Different OSes handled the logical memory differently, but as a rule of thumb the kernel runs it's own code in a separate memory space and the task code in it's own memory space. It's responsible for managing both it's own memory and memory of the user tasks. In fact, most OSes run in either: kernel mode or user mode, depending on the routine being executed. Kernel routines and functions run in kernel mode. While the middleware and application software runs in the user mode. If user level code wants to access something in the kernel, they must do so via system calls, theses system calls act

as an abstraction interface between user space processes and the kernel subroutines.

I/O management

Like memory, the embedded OSes also have to provide an abstraction between the device drivers and the higher level application software. The OS is responsible for providing a universal and a uniform interface for I/O devices that perform a multitude of different operations via system calls. Doing so will allow it to fairly and efficiently manage the I/O among various user space tasks at the same time and the same time protecting the I/O devices as the devices are only accessible via the system call interface. It also has to manage the synchronous and asynchronous communication to and from the I/O devices and the user tasks.



To achieve this, the I/O management scheme typically constitutes of a generic device-driver interface both user processes and device drivers as well as some caching mechanism employed in the middle. The device driver are on the other hand responsible for controlling the actual device on board. Based on the platform, each device driver must implement a specific set of function such as startup, shutdown, enable, disable et al. This varies per platform and usually tightly controlled by commercial embedded OSes. As mentioned earlier the OSes provide an API which control how the higher level processes talk with the device drivers. For the processes higher up in the OS model, these devices act as black boxes and the only method for accessing them is via the API specified by the kernel.

Timer services

Along with a host of interfaces and functionalities mentioned above the real time OS must also provide the developers will clocks and time services. The time service provided by the operating system is called the system clock. The system clock is maintained by the kernel based on interrupts from a hardware clock. The tricky part here is the time resolution; based on the current technology the hardware clock resolution is somewhere around 1 nanosecond (considering processor clock speed to be 3GHz). But this much fine resolution is not being made available to the user, the user processes get the time with resolution in milliseconds. This discrepancy in time is primarily due to the overhead associated with the hardware clock interrupt ISR, which

is the amount of time the processor spends on setting the system clock from the interrupt received from the hardware clock. Another problem is the jitter associated with preemptive task switching which sometimes leads to error in the accuracy of time.

There are many solutions for providing high resolution timer services, one of the easier one to implement is to write the current timestamp in the memory accessible by an application software. One such example is a Pentium processor, a user thread can be read the Pentium time stamp counter. This counter resets to 0 when the system restarts and increments at every processor cycle. Based on today's processor speeds this counter increments several times during a nanosecond. Mind well this is a platform specific implementation and it will be difficult to port the application which uses this facility to a platform which may have a different resolution clock or may not have this facility at all.

Along with the clock services the operating system also provides programmers with two types of timers - periodic timers and aperiodic(one shot) timers.

Periodic timers

Periodic timers are used to keep track of repetitive tasks like polling or performing some activity periodically. After a periodic timer is set, each time it expires a handler routine is called and the timer gets re-inserted into the timer queue.

Aperiodic (one shot) timers

These timers are set to expire only once. One popular example of an one-shot timer is the watchdog or the grenade timer. Watchdog timers are used extensively in real time systems, they form a part of the failsafe mechanism which is activated as soon as the timer expires. It is usually used to handle deadline misses and calling the exception handling routine when the timer expires.

Common RTOSes

We'll discuss some of the more prevalent operating systems available commercially in the markets today.

pSOS

Supported Platform: Motorola 68000

Kernel Size: 12K

The original embedded OS. It was first developed in 1982 by SCG. Currently being supported by NXP Semiconductor through TriMedia VLIW core which they brought from Wind River Systems. pSOS supports priority inheritance and priority ceiling. It also supports segmented memory management. An example application of pSOS is a base station in a cellular network.

vxWorks

Supported Platform: x86, x86-64, MIPS, PowerPC, SH-4, ARM, SPARC Version 8 (V8)

This operating system has been to Mars, twice. It was used in both Pathfinder and Curiosity mission. It comes with its own Eclipse-based IDE called Tornado. Development is carried out using the host-target paradigm the developers can use the superior facilities of developing on a 'host' system like an editor, a compiler toolchain, a debugger, and an emulator. The software is then compiled to the 'target' system.

VRTX

Supported Platform: ARM, MIPS, PowerPC, RISC

Kernel Size (for VRTXmc): 4K to 8K ROM and 1K RAM

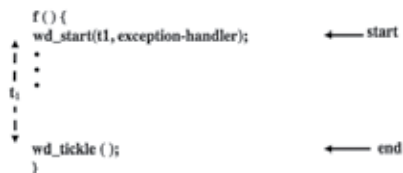
VRTX is developed by Mentor Graphics. It comes in two versions: VRTXmc (micro-controller) for small systems requiring minimal memory use and VRTXsa (scalable architecture) for full operating system features. VRTX powers the Hubble Space Telescope it has also been by the US FAA (Federal Aviation Authority) for use in avionics. VRTXmc was used extensively by Motorola in many of its mobile phones.

Windows CE

Supported Platform: x86, MIPS, ARM, SuperH

Kernel Size: 350K

Windows CE currently billed as Windows Embedded Compact is the embedded offering from the PC giant. Microsoft directly licenses Windows CE to OEM and manufacturers they can modify the user interface while Windows CE provides the technological backbone. It provides 256 priority levels. To optimise performance, all the threads run in the kernel mode.



Use of one shot timers in functions with critical section

C/OS-II

Supported Platform: ARM Cortex-M3, ARM Cortex-M4F, ARM ARM7TDMI, Atmel AVR

Kernel Size: 20K

MicroC/OS-II is a free for non-commercial use developed by Jean J. Labrosse for a two-part article in Embedded Systems Programming magazine, much like Andrew Tanenbaum who created Minix when writing a textbook. μ C/OS-II has been ported to over 100 different platforms ranging from 8-bit to 64-bit microprocessor, microcontrollers and DSPs. This OS is also vetted by the US FAA for use in avionics.

RTLinux

Supported Platform: All hardware supported by Linux

Kernel Size: 125-256K

RTLinux runs along with a stock Linux system. The real-time kernel sits between the hardware and the actual Linux system. The RT kernel acts as a proxy and intercepts all the interrupts from the hardware. If an interrupt causes a real-time task to run, the real time kernel preempts Linux and lets the real time task run its course. In a way, Linux itself runs as a task of the real time system. The real time tasks are loaded as kernel modules, and thus they're part of the kernel and run in kernel space. It is used to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines.

QNX

Supported Platform: Intel 8088, x86, MIPS, PowerPC, SH-4, ARM, StrongARM, XScale

Kernel Size: 12K

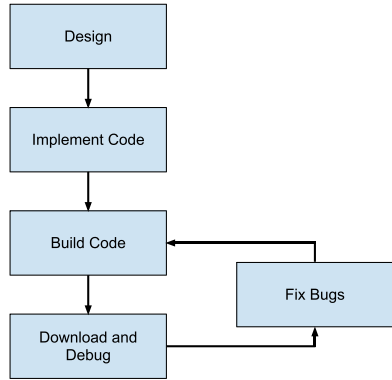
QNX has a long and colorful history. In its most current avatar it's being called Blackberry 10 is running Blackberry's newest smartphones. The microkernel architecture offers immense advantage as the ability to finely tune the OS to your exact specification is immensely beneficial to manufacturers producing components in large volume, where even 1% reduction in memory can reduce the cost by millions of dollars.

PROGRAMMING EMBEDDED SYSTEMS

This chapter deals with programming and development on embedded systems. We'll be touching upon the lifecycle of an embedded project, hardware, cross-compiling and toolchains, debugging and emulation, RTOSes, tracing and math workbenches to be utilized when working with DSPs.

Overview and SDLC

Developing for an embedded system is an entirely different ball game as compared to a traditional non-embedded system. This is due to the constraints imposed by these systems that we mentioned in the previous chapter - hardware, resources and time. A program has to be usually customised for the platform that has to be deployed on i.e. cross-compiled for each of



the platforms you want it to work on. Another thing to consider is that you'll also have very limited functionality in terms of what one refers to as a standard library, only the most important functions find their way there. There may also be a limitation in the memory which affects the size of program that you can deploy to most systems.

And of course, time, the ever present elephant in the room – some tasks will have a hard-deadline i.e. they have to be completed on or before the deadline after that the task has failed even if the result is logically correct. Say for example you're making a smart electricity meter, one with a GSM module baked in on the board which periodically sends the electricity usage and at the same time take and execute actions based on commands from a command and control server. For such a system, the remote command takes the priority above everything else, say for example if the supervisor sends a command to switch off the power supply – the result should be instantaneous, this task takes precedence above every other task that the system is performing. You will have to make design decisions and write your software in such a way that such systems can run reliably, in all conditions, as once they've been deployed it's difficult to make modifications in such systems.

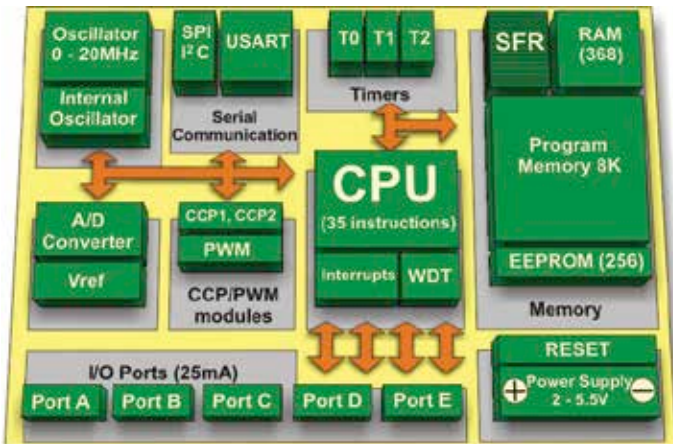
The life-cycle for a typical embedded software looks something like the figure shown below. You start with designing the system at a conceptual level after having collected all your requirements and having clearly defined your project goals. After this you write your code in the language of your choice supported by the platform. This is followed by building your code, this is where the compiler comes in, the compiler(s) that you use for your project will take the code and create an executable which depends on your platform can be pure assembly or optimised bytecode. After this you deploy your

code on your board, now how you do this will again change from platform to platform but for development purposes most boards provide a 'host-target' interface which allows the user to use a normal *nix/Windows machine and the development version of the board is attached to the host via USB, ethernet, parallel port or RS-232 and the developer can directly download the code into the board and run it there. This changes for production systems where the code is usually burnt into a programmable ROM, which can be easily mass produced.

Another important aspect is debugging, this is where in case of errors in your program check the status of the system step-by-step and zero in on the problem area. The debugging methods also vary from platform to platform, we'll discuss them in detail later. The programmer then fixes the bug builds the program and deploys it again checking for errors and thus the cycle continues, until the code is ready for production.

Hardware

The first thing that you as a software engineer on an embedded project should do is to get yourself familiarised with the hardware on which the software will run. We can't stress this fact enough, each board has it's own idiosyncrasies, hardware components and some feature that are entirely unique to that board like the timestamp counter and the MMX registers in the Pentium processors which are not available on other Intel boards, let



PIC16F887 Microcontroller block diagram

alone boards from other manufacturers. Initially you only need to understand the basic operation of the system – the main purpose of the board, how the data flows through the board and the input/outputs available.

Once you're clear about the basics it will time to dig deeper. You must then go through the hardware specification which contain detailed schematic of components and peripheral available in the board. The best way to start is to go through the 'user guide' or 'programmers manual' that came along with the board. This usually contains a detailed description of each component/peripheral on the board, it's hardware specifications and how it can be programmed. In case the board is custom designed for you then you'll find a more pedantic documentation may be geared towards hardware engineers. The next step is understand how memory is managed by the system. Processors use a set of two buses to interact with the memory - the address bus and the data bus. To read or write to an address the processor selects an address on the address bus which is then passed to the address decoder which translates the location's logical address to the physical address. There is also a control bus which is used to control the bus signals like read, write and chip-select/chip-enable(CS/CE).

Next step is getting to know the processor. This will not take much time if you're going to do all your programming in a high-level language like C. You'll have to take some time and understand how the peripherals of your processor work. If you're using C, most processors will look and act pretty much in the same way. However, if you're planning to program them using assembly, then it's advised that your familiarise yourself with the processor's register architecture and it's instruction set. Once again the databook will be your best friend here, processor user guides/databooks are usually well written as databooks go.

The final step as far as understanding the hardware is concerned is to understand the working of all peripherals and devices that will be used in your system. These devices communicate with the processor by interrupts and I/O or memory-mapped registers. Depending on your application, the device(s) might include LCD or keyboard controllers, analog-to-digital (A/D) converters, network interface chips, or custom application-specific integrated circuits (ASICs). Procure a data-sheet for each component, in the early stages of the project you only need to understand the basics of each of them.

A good programming practice is to collect all the data sheets/user manuals in one place in either physical or electronic form. This way you can

keep it handy with yourself when you're actually programming the board and can look up something very easily.

Programming languages

The language you can use to write your program will vary from board to board. These section will list some of the major languages that you can use to write your programs.

Assembly

Assembly language is a low-level programming language. Instructions in assembly language also known as a mnemonic, usually maps one-to-one to the platform's machine code instructions which is collectively called it's instruction set. One of the drawback of assembly language is that the language is specific to an architecture, which is quite the opposite of portable high-level languages.

An assembly program is usually assembled by a program called an assembler. It essentially performs the act of converting assembly code to machine code. Assembly language is usually used in smaller projects which must have their softwares execute without a run-time components or libraries associated with a high-level languages. Examples of such systems are firmware for telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors. Programmers usually prefer high-level language like C to write their programs. Such languages abstract the details of the actual implementation from the programmers giving them a consistent and easy-to-use interface for writing programs. However some tasks can be performed better in assembly and some can performed only in assembly.

The following is an assembly program written for an Atmel AtMega16 microcontroller. It is written to solve the equation: $c = x^2 + 2xy + y^2$

```
.include <m16def.inc>           ; description file for an AtMega16
.def temp = R17                 ; alias for register R17
.def x = R18                    ; alias for register R18
.def y = R19                     ; alias for register R19

    LDI x, 0x0A                 ; x = 10
    LDI y, C                     ; y = 12
```

```

    MUL x, x                ; R1:RO = x × x
    MOV temp, RO           ; temp  RO
    MUL x, y                ; R1:RO = x × y
    LSL RO                 ; RO = 2 × RO
    ADD temp, RO           ; temp = temp + RO
    MUL y, y                ; R1:RO = y × y
    ADD temp, RO           ; temp = temp + RO

```

```

loop:
    RJMP loop              ; loop infinitely

```

Assembly code is then translated to machine-level instructions by the assembler. It will look something like this:

```

Address
Mach. Language Instruction
+00000000:
E015 LDI R18, 0x05
+00000001:
E026 LDI R19, 0x06
+00000002:
9F11 MUL R18, R18
+00000003:
2D00 MOV R17, RO
+00000004:
9F12 MUL R18, R18
+00000005:
0C00 LSL RO
+00000006:
0D00 ADD R17, RO
+00000007:
9F22 MUL R19, R19
+00000008:
0D00 ADD R17, RO
@00000009:

+0000000A:
CFFF RJMP PC-0x0000

```

Assembly as you can see from the program can become quite cumbersome to manage as the complexity of the project increases. One of the important feature of a programming language is that it should be readable, a long assembly program tends to be so complex that it can be difficult even for the person who coded the whole thing in first place to understand the code.

C

C is perhaps the most popular language for programming embedded systems. The reason for this being the fact that C can be considered to a middle-level language. It is much more readable than vanilla assembly and provides developers with powerful features which can make the programming easier. At the same time, C is known to provide a low-level access to the hardware which makes it a favorite among the programmers as it brings together best of both worlds into one package. Another main reason for C compilers being so famous is that the C compilers are available for a wide variety of platforms. These compilers have seen a major boost in performance in recent years, and depending on whom you ask, C nowadays is considered to be as fast as assembly. Portability is also a factor here as there are a host of cross compilers out there which take a source file in C and can create the executable for any platform and architecture.

Consider this code snippet for example, which resets bits for Port H on a Motorola 68HC12

```
PORTH &= 0xF5; // Changes bits 1 and 3 to zeros using C
PORTH &= ~0x0A; // Same as above but using inverting the bit mask
// which makes it easier to see which bits are
// cleared
```

This would translate to something like this in assembly:

```
BCLR PORTH,$0A ;Changes bits 1 and 3 to zeros for Motorola 68HC12
```

You can see how easy it is to manipulate the low level hardware from directly inside C. This makes it the go to language for most platforms out there. There are a couple of features that we'd like to highlight.

Bitfields

Bitfields allow the programmers to access memory in unaligned sections, and even in sections smaller than a byte. In C, a bit field is created using a struct.

```

struct _bitfield {
    flagA : 1;
    flagB : 1;
    nybbA : 4;
    byteA : 8;
}

```

The numbers in the structure above are the field's size in bit's and not bytes. Here both flagA and flagB are 1-bit long so they're boolean variables. nybbA can store at most 4 bits so it can take a value to 0xFF (15 in Hexadecimal)

Fields in a bitfield can be address like a regular structure

```

struct _bitfield field;
field.flagA = 1;
field.flagB = 0;
field.nybbA = 0x0A;
field.byteA = 255;

```

Const

When a variable is declared as const, it means that the value of the variable will not be modified by the function. In embedded system paradigm declaring a variable to be const makes sure that the data in the variable is stored inside the ROM. An example for the first use case would look something like this:

```

void print_string( char const * the_string );

```

This makes it clear that the function will not modify the original string (note that we're passing the variable by reference here). An example for the second use case would be

```

const char * months[] = {
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December",
};

```


Here the content would be directly stored in the ROM and the program will have to look up the ROM in case it wants to access data from this variable.

Volatile

When a programmer declares a variable to be volatile, it signals both the compiler and the developer that the value of this variable can change anytime and this can happen by means which are outside the normal flow of the application. The changes can be caused by a peripheral, another processor in a multiprocessor system or an interrupt. The signal to the compiler in this case is to ask it to not optimise this variable. Usually volatile is used for input and output devices which are stored as variables in C.

Now, let us see (pun not intended!) how we can write the blinking LED program for an ARCOM board. The blinking LED program is the “Hello World!” program of embedded system. It is usually used by the programmers to get a feel of how to program for a given board – it’s features and specialities which they can further use for the task at hand.

```
#include "led.h"

int main(void) {
    /* Configure the green LED control pin. */
    ledInit();

    while (1) {
        /* Toggle the state of the green LED. */
        ledToggle();

        /* Pause for 500 ms */
        delay_ms(500);
    }
    return 0;
}
```

This the main function which controls the program’s flow. We next define the helper functions – ledInit, ledToggle() and delay_ms().

Given that there is a PXA255 Processor on the board, we next have to go through the developer documentation to find out which pins control the onboard LED. According to the PXA255 Processor Developer’s the

configuration of the GPIO pins for the LEDs are controlled by bits 20 (red), 21 (yellow), and 22 (green) in the 32-bit GPDRO register. These registers are memory mapped, thus they're easily accessible in C. We can read and write to these registers as if we are reading or writing from a memory location. This is what the `ledInit` function would look like.

```
#define PIN22_FUNC_GENERAL    (0xFFFFCFFF)
#define LED_GREEN            (0x00400000)

void ledInit(void) {
    GPIO_O_CLEAR_REG = LED_GREEN;
    GPIO_O_FUN_HI_REG &= PIN22_FUNC_GENERAL;
    GPIO_O_DIRECTION_REG |= LED_GREEN;
}
```

Next up is the `ledToggle` function which turns it on and off.

```
void ledToggle(void) {
    if (GPIO_O_LEVEL_REG & LED_GREEN)
        GPIO_O_CLEAR_REG = LED_GREEN;
    else
        GPIO_O_SET_REG = LED_GREEN;
}
```

And finally to make it blink the delay.

```
#define CYCLES_PER_MS        (9000)

void delay_ms(int milliseconds) {
    long volatile cycles = (milliseconds * CYCLE_PER_MS);
    while (cycles != 0)
        cycles--;
}
```

The four functions `main`, `ledInit`, `ledToggle` and `delay_ms` control the blinking LED program. We'll discuss the building and execution of the program in the next section.

Java

Java when it was initially being developed by SUN was a technology for programming the next generation of smart appliances. We've come a long way since then - and Java has now grown into a programming language that is trusted by legions of programmers across the globe. It's stability and portability have attracted a lot of users, one of the reason why it is the work-horse for enterprise software. The offerings of Java for embedded devices is growing at a tremendous rate. Silicon manufactures are now coming up with embedded Java Virtual Machines eg. the ARM Cortex M-Series which requires only 50K Flash and a few kilobytes of RAM.

The 'write once, run anywhere' approach that Java follows makes it very important and an effective language for embedded systems. As long as there is a JVM for the platform to which it's being deployed, it's guaranteed that your code will run on that machine. Java for embedded systems comes in two flavors:

Java SE Embedded

Java Standard Edition embedded is based on the desktop Java SE. It is designed to be used on systems with at least 32MB of RAM. It can work on Linux ARM, x86, PPC, Windows and Windows CE architectures and platforms.

Java ME Embedded

Java ME embedded is based on the Connected Device Configuration which is a subset of Java Micro Edition. It's designed to be used on systems with at least 8 MB of RAM. It can work on Linux ARM, PPC and MIPS architecture.

Python

Python too can be used on embedded systems. How exactly you can use Python is governed by the limitations of the hardware. Some of the modern embedded devices have enough memory and processor to run a typical-Linux environment. So getting CPython to run on such systems is just a question of cross-compiling and tuning the performance. The devices which can run CPython in are: Gumstix, FIC Neo1973 and Neo FreeRunner and the Telit GSM/GPRS modules.

In this case when we say Python we mean the Python interpreter. For programming the board using python you only need to deploy the Python file to the device where the interpreter will execute the script. In a way this can be considered similar to the JVM.

Some devices won't be able to accommodate the entire CPython inside their memory. In these cases a modified or adapted version of Python needs to be used. Examples of such implementations are PyMite and Tiny Python. These versions of Python are so heavily reengineered that you can even get away with calling them a different version of Python. PyMite for eg., is a flyweight Python interpreter which can execute even on 8-bit microcontrollers with resources as limited as 64K Flash and just 4K of RAM. PyMite supports a subset of the Python 2.5 syntax and can execute a subset of the Python 2.5 bytecode.

Other Languages

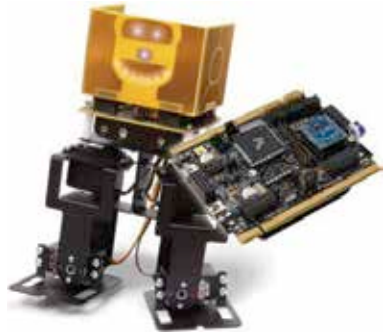
Apart from these languages there are a lot of other languages which are used by some for programming embedded systems. The more important ones that we'll discuss are Forth, Ada and Basic.

Forth is a stack-based language and a software development environment. The language much like Python is useless without its interpreter. Working with Forth is different compared to other programming languages, it works in a more 'hacky' way. You program with Forth as if you're making a complex shell script. You write a little, test a little and repeat the process till you get the result. Forth is an interactive language so it makes it very easy to interact with the target system. Forth's extensibility is also worth a mention. Forth lets the developer extend and control the compiler itself, something which we cannot do in C. Forth is currently used in is currently used in boot loaders such as Open Firmware, space applications and other embedded systems.

Ada is other languages that has achieved some measure of success in the embedded system market. Ada was conceived to work exclusively on embedded systems. Developed by DoD, the focus when developing the language was the safety and reliability of the systems designed using Ada. Ada compiler go through rigorous checkings against the exhaustive ACATS (Ada Conformity Assessment Test Suite), because of these safety features it finds extensive usage in military, aviation and air traffic control, commercial rockets, satellites and other space systems.

And finally BASIC. You may be surprised to find this language here, as it has always considered to be something of a toy language to learn programming. But Basic finds serious usage in embedded system. Basic's simplicity is its greatest strength – it is very easy to learn. It runs at roughly 95,000 instructions per second. A lot of boards come

with a Basic interpreter which can run the Basic code. The programming is very easy too, consider the following code used to read the accelerometer values from an MMA7455 accelerometer on a Freescale Bipedal Mechatronics Robot based on the 32-bit ColdFire MCF52259. This runs a version of StickOS which comes with a Basic interpreter:



Freescale Mechatronics Robo running StickOS

```

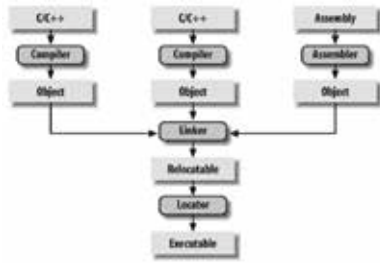
5 rem tilt_demo
10 dim x, y, z
20 gosub mma7455_init 0x1d // library function to initialise
// MMA7455
30 while 1 do // infinite loop
40 gosub mma7455_poll 0x1d, x, y, z // library function to read
// MMA7455
50 print hex "x = ", x, "y = ", y, "z = ", z
60 sleep 100 ms
70 endwhile
  
```

Building Software

This is the most crucial section of this chapter. In this section we'll discuss how to run the software that you've written on the target system. This part applies to the programs written in C and Ada. The other programming languages that we mentioned are interpreted except Assembly which is directly converted to machine code. The usual workflow when developing for an embedded system is:

- Create a C file(s) on the host which contain all your functions.
- Compile/Assemble the machine code on the host
- Link all the object files and libraries and resolve all symbols on host
- Assign memory addresses to the code and data on host
- Download the executable image onto the target processor memory
- Reset the target processor
- Voila! Your program is working.

In this section we'll talk about the compilers, linkers and assemblers along with build systems which will handle all these tasks transparently for you with the only requirement being the configuration. Downloading the executable to the target system is usually carried out through either USB, serial IO or network. The host requires the target boards interface drivers to be present so that it can send data to the board. The interface is then responsible for moving the code and data to the correct place in the memory.



A generic embedded software development process

Cross Compilers

Microcontrollers do not have a full-fledged operating systems which can run compilers for the code for that particular platform. This is where cross compilers come in, a cross compiler is capable of creating executable code for a target platform which is different from the platform on which it is running. Eg. you're using your x86-64 based system to generate code for an ARM processor.

There are a multitude of cross compilers available in the market today. One of them is GCC, a collection of compilers from GNU. GCC can be setup to cross compile to a different platform. GCC needs a compiled executable of binutils to be made available for each of the target platform. Another important part is the GNU Assembler. binutils must be compiled with the `--target` switch set to the desired target platform in the configure script. GCC also has to be configured with the same target option. Cross compiling GCC requires that at least a subset of the target's C standard library be available on the host. You can choose to have the full C library but a better alternative is to use newlib, which is a smaller, light weight C library containing only the important modules required to compile the C source code that you provide.

Another such compiler is ELLCC which uses Clang and LLVM compiler infrastructure to provide an easy to use multi-target cross compilation environment. It requires ecc, the ELLCC C/C++ compiler, binutils, libecc and QEMU for working. GNU Gnat is a compiler for Ada, it is under development and has an engaging community.

Linkers

A linker or link editor is program that takes the objects generated by the compiler and assembles them into a single executable program or a library. The result of compiling a source file is an object file. Compiling a project can result in a number of object files. The object files themselves are incomplete, as some of the internal variable and function reference to an external library are still unresolved.

Linker is responsible for combining these object files and to resolve all the unresolved symbols in the object file(s). These unresolved objects are acquired by the linker from a library, eg. libc or the Standard C Library we mentioned earlier. For embedded systems it is advised that we use a subset of the libc something like newlib. Once all the unresolved objects have been resolved to ones in a given library a linker's work is done. There is still one more step for embedded systems though, memory allocation. The generated code must be provided with memory addresses for loading the code and data in the target processor memory. You'll have to provide the information about memory to a program called the locator. The locator uses this information to assign physical memory addresses to each of the code and data sections. The locator produces an output file that contains a binary image which can be directly loaded into the target ROM. A commonly used linker/locator for embedded systems is GNU ld.

Build Systems

Building a software for just one target system is fairly easy to manage. You just need a proper project structure in place and properly configured compiler and linker/loader toolchain. If you plan on developing and deploying to multiple systems at the same time, you'll be better off with a build system. Build systems like the few mentioned below provide an easy way to manage the build process for your programs. These systems only require some configuration for your target platform, it will manage the rest of the nitty-gritties by itself.

Buildroot <http://www.buildroot.net/>

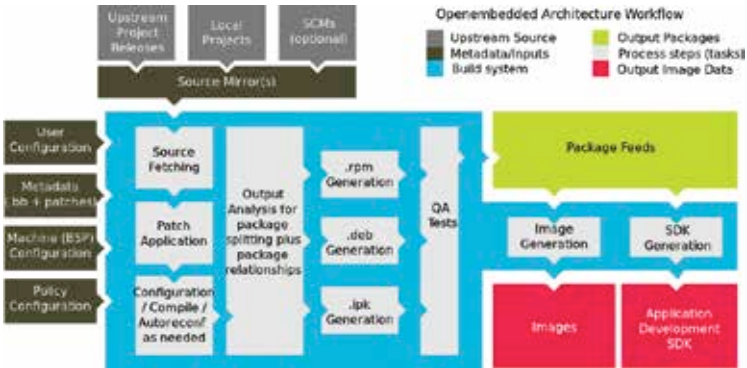
Buildroot-ng <http://wiki.openwrt.org/>

crosstool-NG <http://www.crosstool-ng.org/>

PTXdist <http://www.ptxdist.org/>

OpenEmbedded <http://www.openembedded.org/>

OE-lite <http://oe-lite.org/>



Yocto project build system schematic

muddle <https://code.google.com/p/muddle/>

Poky <http://pokylinux.org/>

OpenBricks <http://www.openbricks.org/>

Yocto Project <http://www.yoctoproject.org/>

Scratchbox <http://www.scratchbox.org/>

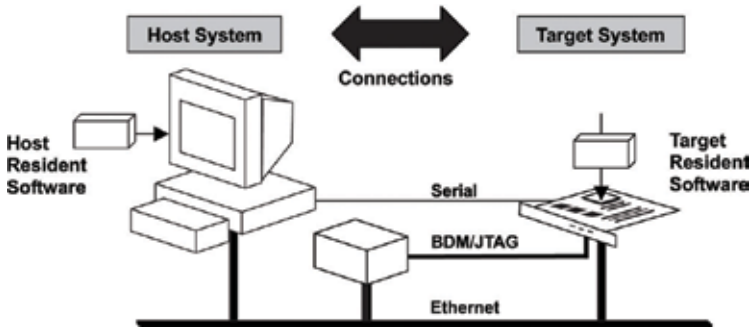
Cross Linux From Scratch <http://www.cross-lfs.org/>

Aboriginal Linux <http://landley.net/aboriginal/>

We cannot discuss each and every build system here but for the sake of example consider the Yocto project. Yocto project is specifically geared towards building custom Linux images for your target platform. It is based on the Poky project, which it uses to construct complete Linux images. The Yocto project utilising the OpenEmbedded build system provides an open source development environment which targets ARM, MIPS, PowerPC and x86 architectures for a variety of platforms including x86-64 and emulated ones.

Debugging

This is perhaps the most important part of the entire workflow and particularly indispensable in the earlier in the software development timeline. Those of you who have some experience in this matter on developing software on non-embedded systems, will know that debugging involves running the code step-by-step usually single line at a time to find out a problem with the program. Since we cannot even run general programs on an embedded system, running a full fledged debugger is out of question, this makes debugging your program very difficult.



Embedded debugging schematic

Mind well, the real world and real time constraints apply here too, think of it like this, say you stop on a breakpoint where your system is controlling a servo now if you stop at that particular point chances are that you'll permanently damage the motor. Hence, developers extensively use serial IO channels and error-message style debugging in the embedded systems paradigm.

Debugging a microprocessor centric embedded system is different from debugging an embedded system where processing is performed by DSP, FPGA or a co-processor. Today, a large number of embedded systems have started using processors have more than one core. A common problem that developers face is the proper synchronisation of software execution while debugging. In such cases the system will have to implement low-level debugging on the busses between the processor cores. Thus the debugging strategies vary from system to system. Following are some of the methods involving both hardware and software which can be used to debug an embedded system.

Resident Debugger

A resident debugger functions as a normal debugger in our traditional development paradigm. The program runs directly on the microcontroller itself, while the developer is able to control the actions from the host machine. A resident debugger usually occupies some resources from the target machine including a communication port, an interrupt to handle single stepping and some memory for the resident module of the debugger.

External Debuggers

We mentioned earlier that programmers developing in this paradigm rely on serial IO channels and error-message style debugging. This debugger

relies heavily on logging or outputs on serial ports using either a monitor in the ROM or using a debug server like the Remedy Debugger. Note that this is not what we call debugging in our usual sense, the programmer cannot step through the code line by line, rather the output of each line of execution is printed out to the serial port or a flash memory, the user then has to manually go through the code results to find out what exactly is going wrong. Another example of an external debugger would be the ADB (Android Debugging Bridge) used in Android smartphones. You start an ADB server on your host and connect to the target i.e. the phone using a USB-cable to the system. You can then debug the code on your smartphone usually an application from the host. Hackers utilise ADB to inject exploits into the phone which help bypass the restrictions on the bootloader and root the phone.

In Circuit Emulators

An In Circuit emulator is a speciality hardware device, it is developed with what is known as a bond-out processor. These processors have internal signals and buses brought out to external pins for monitoring. These signals provide the developer with the information about the current state of the processor. An ICE replaces the target chip and acts like



RTE870_C In circuit emulator

the processor to the rest of the circuit but it comes with a host of internal mechanism which allow the developer to see what's going on inside, set breakpoints, load new code, grab traces etc.

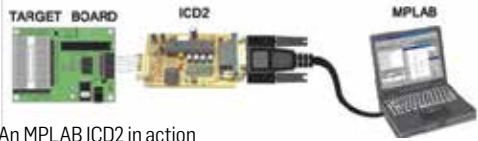
The problem is the cost associated with this kind of emulator is the cost of manufacturing a bond-out version of the target chip, for the current processors the speed have increased so much that it is nigh impossible to take anything off the chip. Another problem is that the ICE requires the chip (i.e. the ICE chip) to be in socket, or requires a special adapter mounted in the target chip socket so that the ICE can connect to it's lines.

For this particular reason recently JTAG (Joint Test Action Group) based hardware debuggers are being considered to be ICEs. This JTAG based hardware provides access equivalent to ICEs using on-chip hardware even

on standard production chips. This greatly decreases the cost associated with production of ICEs, though the name ICE doesn't fit the bill and is actually a misnomer as there is not emulation taking place in this case.

In Circuit Debugger (ICD)

An In Circuit Debugger on the other hand uses special debug hardware added to the target chips which try to provide the developer with ICE-like capabilities. The advantage that ICDs offer is that the code runs on the target chip and not something that emulates the target chip. The cons of



An MPLAB ICD2 in action

using the ICDs are firstly, they require an access to debugging lines which often have multiple roles. You can't use those pins in other roles during a debugging session. ICD also involves adding extra circuitry to the chip, and as the size and cost are a limit many compromises are made on the features available with a chip



MPLAB ICD3

for eg. tracing which will require a larger RAM for buffering which may not be available. Though these are not much of a problem on larger chips were a few lines dedicated for debugging is not much of an issue but it can severely undermine the performance in a smaller chip.

Emulators

Emulators abstract all the aspects of the hardware and provide them to the user in the development environment. It give the developer the full control of the hardware. This does away with the cost involved in using hardware debuggers but the major downside is that of the speed of operation, the system can sometime be 100x slower than the actual processor.

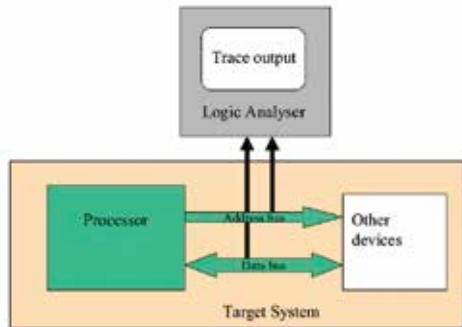
Tracing

This may as well considered to be a part of debugging but tracing is much different from traditional start and stop debugging thus we're discussing the topic separately. The problem with traditional start/stop debugging technique is that it will not work for a system with real-time constraints. A debugger actually stops the processor as soon as it hits the breakpoint, this

has two major problems in real time systems:

If the code was in a critical region accessing some hardware, it can crash the hardware as we mentioned earlier.

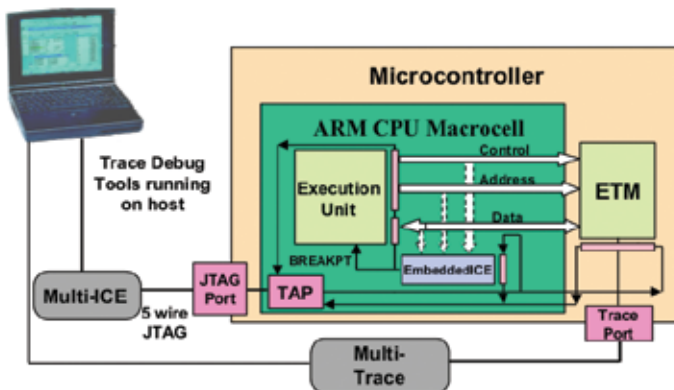
Even if it is safe to stop the processor, some tasks may have hard real time requirements which can fail if you stop the processor.



MPLAB ICD3

This is where the tracing comes in, real-time trace captures a trace (a series of) instructions that are executed by the processor in a buffer which can be analysed later. In addition to the instructions executed the data that is used by those instructions is also stored. It is also possible to select a trigger condition for a trace to begin – execution of a particular instruction, writing of a particular value, writing to a particular location in RAM et al. You can also construct complex triggering conditions by mixing together various other condition.

To procure a real time trace of the system on a trigger, one needs a logic analyser. The logic analyser is connected to the processors address and



MPLAB ICD3

data busses and it writes all the data going through it to the memory both the instructions and the data. This is then used to construct the sequence of code that was executed by the processor when the trace condition was triggered. The problem in this case is that such logical analysers are very costly and more than that such analysers may not be able to access the busses on modern processors. Also modern processors use various levels of caching to speed up execution, they also prefetch data from the memory out of order for optimising the performance which makes it very difficult to create a trace of the statements executed by the processor. Modern processor now come equipped with an embedded trace macrocell (ETM) which is essentially a logical analyser that is baked right into the board.

FUTURE OF EMBEDDED SYSTEMS AND CAREER OPTIONS

Embedded systems have come a long way since their inception. Today, some toilets and toasters can tweet about what they're upto. From smart clothing to smart banking, embedded systems have accentuated technology's growth by manifold

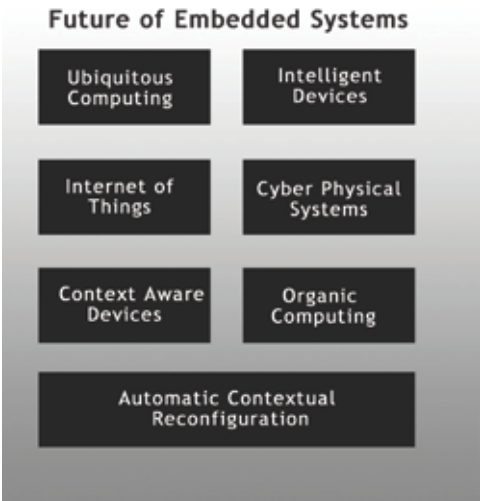
With growth and advancements in the field of electronics, wireless communications, networking, cognitive and affective computing and robotics, devices around you communicate in more ways than you ever imagined. Those times are not very distant when every object around us will have a small processor/sensor embedded within itself, invisible to us but still

communicating with all other devices around, making our lives more connected and accessible than ever before.

The future of embedded systems lies in the advancement of technologies that enable faster communications, heavy data storage capacities and highly interwoven connections among the devices. Before diving into the enormous number of applications of embedded systems, let’s discuss the seven buzzwords that will define the future of embedded systems.

Ubiquitous computing

Ubiquitous Computing is a branch of computing that focuses on interconnected and communicating devices carefully integrated into the objects we interact with in our daily lives. These objects can be anything right from your clothes to your toasters and coffee mugs. Smartphones and tablets are currently the obvious targets for applications aiming at ubiquitous computing, but in the future, don’t be surprised if your game console talks to your smartphone’s calendar about how “busy” you are today. The term Ubiquitous Computing (Ubicomp), also known as Pervasive Computing, was coined around 1988 by Mark Weiser when he was heading the Xerox Palo Alto Research Center (PARC). Sentient Computing is another form of ubiquitous computing which involves using various kinds of sensors to sense the environment and react accordingly. All over the world, research



The future of embedded systems

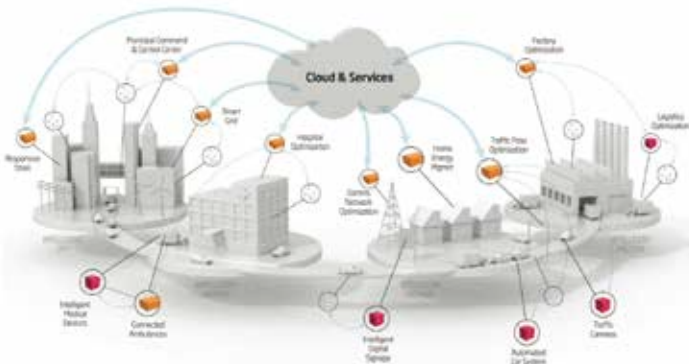
institutions and universities are working on Ubicomp's possibilities, bringing us more and more applications of everyday devices communicating with each other. With advancement in cloud computing, applications of ubiquitous computing got a real boost. Apple's iCloud is a very basic implementation of ubiquitous computing which integrates all your Apple devices seamlessly and maintains unified data among all those devices.



iCloud : A basic implementation of ubiquitous computing

Intelligent devices

Intelligent devices are devices that have the ability to think, or simply put are “things that think”. These devices use a combination of technology, algorithms and embedded hardware to replicate what was once thought to be an activity exclusive to living beings with brains. Today, there are machines that can do your thinking for you. MIT Media Lab is working on this “Things that think” idea and aims at creating environments that enable this way of thinking. Whether it’s a simple device such as iLumi (an intelligent light bulb that can be operated via an Android app to create different lighting environments) or a device as complex as PETMAN (an



Intelligent Systems

anthropomorphic robot that can detect any chemical leaks in its costume – useful for testing chemical protection clothing), intelligent devices will soon be everywhere. This ultra-high level of intelligence in machines poses concerns of security and privacy.

Internet of Things

The Internet of Things is a technology revolution that began just a few years ago. It's gradually sneaking into our lives and will soon be a reality. Introduced by Kevin Ashton from Procter & Gamble in 1989, Internet of Things or IoT is a concept that involves connecting the internet to physical devices such as home appliances and manufacturing machines. With cloud computing and increasing access to fast-



Internet of Things

speed internet everywhere around the world, the Internet of Things will soon be more than just a concept. IOT is no more a discussion with platforms such as Cosm that allow data and devices to be connected in all new ways and OSes such as Contiki that are dedicated to developing apps that realise the concept. Your smartphone communicates with your chair about your sitting posture, with satellite receivers to know the right temperature and your cooking gas to know whether the dish you left to simmer on it is burning; in case of a fire hazard, it will communicate with the respective fire control agency in your zone. Businesses have realised the importance of IoT and there are consultancy firms already that specialise in helping you apply IoT at your organisation.

Cyber physical systems

Cyber physical systems form an important part of Internet of Things. These systems are backed by powerful computation and fast communication and aim at integrating the physical and cyber world into one. Cyber physical systems can be used for precision-based tasks such as in the implementation of robotic arms, exploration-based tasks in areas inaccessible to humans, creating and deploying energy efficient systems as well as for easing daily life activities.

These systems have also found application in exploring outer space. Mars exploration rover, Curiosity employs an intelligent cyber physical

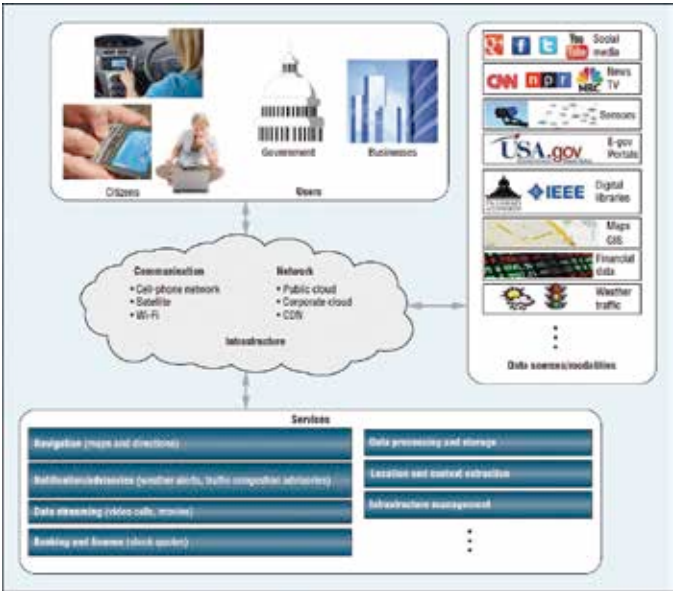
system to automatically navigate the surface of Mars and changes its location according to lighting and weather conditions. Cyber physical systems are challenging to implement as they involve a combination of advanced hardware and software needs. These systems come across issues relating to privacy, security and flexibility due to their high level of complexity.



Mars Curiosity Rover

Context-aware devices

Context awareness, in terms of computing, was introduced by Bill Schilit in 1994. Anind Dey from Carnegie Mellon University defines context as any information related to the situation of an object at an instant of time. Context awareness is an added layer of intelligence to ubiquitous computing and aims at making devices more aware of their surrounding environments. Context-aware devices provide ways by which businesses can understand



Context Awareness Framework

their customers better by facilitating an observable context, knowledge about the world around and social values. Sensing is a crucial phenomenon associated with context awareness. Sensing can be external where visual tracking and location systems are used to collect context or internal wherein sensors are embedded within the user devices and generate context-rich information. Consider a scenario where a doctor has a device that senses its proximity to a particular patient and shows the patient records automatically, saving time for search and retrieval of such information. Or a device that automatically alerts you when it's about to rain and covers your clothes drying in the sun by a shade using humidity sensors. Or a device that detects your mood and changes the lighting of your room accordingly. Great minds are somewhere working in a lab to make all this possible for you.

Automatic contextual reconfiguration

Automatic Contextual Reconfiguration involves sensing the context and reacting in terms of either triggering a task or tweaking the settings of an existing system. Context-aware devices aim at imbibing a strong automatic contextual reconfiguration system. This phenomenon has a wide range of applications. Imagine how amazing it would be if your smartphone could sense when you're entering a meeting room/ICU and automatically put your phone on silent mode. Context reconfiguration is also very handy in security systems such as an application that disables cameras on all devices in a No Photography Zone. Another usage of context reconfiguration is in manufacturing industries to regulate the operation of machines. One such application is in the paper and pulp industry, where the lime kilns need continuous manual monitoring and adjustment of temperature for higher productivity. With embedded temperature sensors that automatically regulate temperature inside the kiln, it's possible to increase productivity and improve quality of the product by reducing manual intervention to check temperature.

Organic computing

Organic Computing adds an extra layer to context awareness and aims at developing intelligent systems that have self-X properties and systems that react to both, changes in itself (endogenous) and changes in the outside world (exogenous). Self-X properties of such systems include self-optimisation, self-protection, self-healing, self-support, self-configuration etc. Such devices behave independent of manual intervention as they can and use machine

learning algorithms to strengthen their properties. Examples include fault tolerant robots and vehicles that are sent to difficult terrains and outer space. These devices are engineered to react to new situations and reconfigure themselves accordingly. Boston Dynamics created a four-legged robot called Big Dog, which can take decisions while navigating terrains. Also, there's a robot called Chembot which has shape shifting features which allow it to inflate and deflate according to terrains it is navigating. Organic computing is alleged with ethical and security concerns stating that fact that such levels of self-X properties can lead to destructive intelligence.

Applications

With advancements in technology trends discussed in the previous sections, future of embedded systems will come with a lot of cool stuff in its bag. We discuss the applications of future embedded systems in different verticals below:

Smart agriculture

Agriculture is one of the primary fields that requires assistance of something awesome like Embedded systems. There are a lot of complexities involved in the process – a farmer has to understand the climatic conditions, sometimes predict the conditions and change the farming practices accordingly. The farming practices also change according to the soil conditions of that specific plot and hence some computational assistance can help a lot.



With this in mind, scientists have come up with what they call as Precision Farming/Precision Agriculture which basically optimises the whole farm management by maximising the output while keeping the input minimum. Precision Farming is presently being implemented in Kerala by Kerala Agricultural University(KAU) and International Centre for Free and Open Source Software (ICFOSS), where they are trying to setup a technology-assisted system named “Smart Agriculture” which

would provide real-time data about the soil with the help of sensors to a cloud-based platform.

This cloud platform would also take information from satellites and suggest farming practices accordingly after proper data interpretation. The system also aims at giving the farmers – market information, post harvest advices and value addition options. This system in the far future also aims at removing the labour issues by setting up robotic farm equipments like sensor-based sprinklers that would do the farming practices that a labour generally does.



Distributed Robotic Garden

In many major countries also, Precision farming has gained a lot of traction. In Holland, presently Driverless tractors are being developed using Real Time Kinematic and GPS. It is proven to be quite effective and cost efficient for large farmlands

Researchers at MIT have setup a Distributed Robot Garden, which is a garden consisting of Tomato plants that are nurtured by Robots. The robots here water the plants, provide nutrients regularly studying the plants condition and harvest the tomatoes optimally. Each plant is equipped with sensors that update the robots about the plant's status; the whole garden has sensors that gives the map and respective positions of the plants to the robots and these robots move around and act according to the plant's condition. Robots right now are able to predict the state of the fruit, as in, when it would ripe and be ready for harvest and when the plant would require the next round of nutrients. Presently, the garden is open for students to conduct research and make this system better and make it commercially usable by Farmers.

Embedded systems can be also used to manage cattle in a better way. For ages now, managing cattle has been done with the help of fences that bind the cattle in a fixed area. However, this is very restrictive and these are physical fences and generally require a lot of money to be put up. Dr. Anderson (U.S. Dept. of Agriculture) came up with an idea of having virtual fences, instead of actually setting up real ones. This can be pos-

sible by using the Directional Virtual Fencing System, which works by equipping the cattle with a pair of GPS enabled headphones that gives the exact location of the cow to a central location. The system also gives information about the landscape pattern, as to where the grass is green and where not. So, the person monitoring the cow can give leverage to the distance the cow travels making the boundary more flexible and virtual. But, if the cow is going completely out of range, it's direction can be corrected – first by emitting a gentle noise, then followed with a louder noise if the cow doesn't listen, and finally a mild shock to reign in really naughty livestock.

In future, this system would also maintain records of the health status of the cattle and also a record of the reactions of each cow to the stimulus given through the system.

Intelligent transport systems

Imagine transport systems that are smart enough to understand the situation they are in and act accordingly. Reminds you of some sci-fi movie from the 90s? Well, it's not sci-fi anymore. Organisations such as Google, Audi and Toyota are into R&D of intelligent transport systems that can increase the safety and comfort of drivers and are in the plans of releasing them for commercial purposes in near future.

New technologies are also used in Aeroplanes these days, which send data about how each component in the aircraft is doing to a central computer for better tracking and management.

Autonomous vehicles are designed to drive on their own by understanding their surrounding and act accordingly.

Implemented with technologies like GPS, advanced Computer Vision, lidar;

these vehicles can identify the correct paths and lanes in a city and navigate without colliding obstacles and follow proper traffic rules. BMW has been doing R&D since 2005 and is all set to release an autonomous vehicle in 2013-2014. Google has already released some videos of the driverless car system that they have been working on since 2005 and has got some positive reviews.

These autonomous vehicles are also equipped with Anti-Lock Braking



BigDog Robot

System which are triggered when the vehicle is in a potentially dangerous situation. Inter-Vehicular Networks are being established which maintains communication between the vehicles. This reduces the risk of potential accidents, as generally accidents happen due to humans not being aware of road context like on “Blind Turns” and on “Steep roads”. So, the objective is to ultimately make a Context aware Information System (CAIS) which maintains information about the context of roads and paths. These systems will definitely help in the reduction of accident rates as the systems are aimed to have an increased reliability and lower reaction time when compared to humans.



Autonomous vehicle

Institutions in India are also using Embedded systems to solve traffic related issues. Due to the increase in population and congestion and the increase in awareness about vehicular pollution, people now prefer public transport systems over driving their own vehicles especially in Metro cities. IIT Hyderabad is presently researching over a Smart Public Transport Notification System (SPTNS) which helps commuters in choosing buses and routes and notifies them with information such as the expected journey time and traffic jams on routes.

Freerange Cooperative Ltd, a New Zealand-based cooperative is presently developing “InfroStructure: A transport Research Project” that basically is trying to explore how digital media technologies can change the way people use public transport systems. This system provides commuters with



Future Intelligent Transport System Model

real-time information about the estimated travel time in the form of mobile games. For this system to work, the whole building has to be seamlessly integrated with new age digital technologies; hence, the floors and walls would consist of sensors and other kind of chips. This system also boasts of a “Shame Detection SubSystem” that helps the transport administration to find out who is travelling without a ticket. Anyone who is in the building without a ticket would be recognised with the help of Radio Frequency Identification Detectors on the floor and a red light will be flagged below the person!

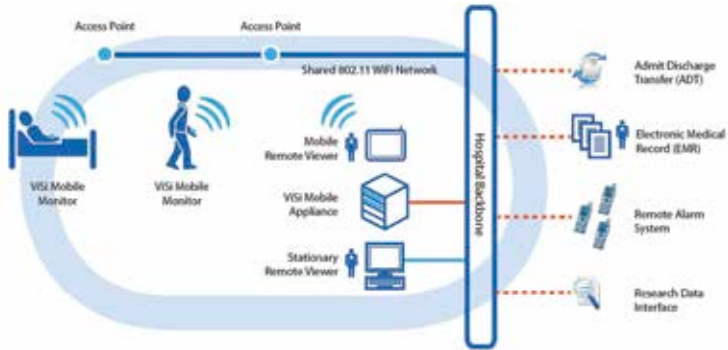
MIT is also developing a system named “CarTel” where cars and taxis act as ubiquitous mobile sensors, communicate with each other and avoid traffic hassles. Presently, maintaining traffic and road data is generally done by helicopters or updates from news; CarTel in a way would revolutionise this. It is presently implemented in some cars in Boston that help drivers avoid junctions with traffic and also alert them if there are any potential problems in the engine. CarTel also can maintain a database of where all the car traveled in a day which can be quite useful for taxi provider organisations.

The Internet of Things can also improve the car-sharing systems that are provided by organisations such as ZipShare. But, presently these car-sharing services are majorly organised around where the car owner lives and not where the car is majorly parked. Majority of the car owners park their cars in their office parking lots or may be in airports and if someone needs a ride from around that spot, the present car-sharing services won't be able to identify that. But, with real-time motion sensors embedded in the car, it would be easier to track cars and also connect them to potential renters.

Smart healthcare systems

Having smart healthcare systems might be one of the most important applications of future embedded systems. It can revolutionise the way medication is done right now - Doctors can provide remote medication to people living in rural areas which is estimated to reduce more than 15 per cent of medical expenditure in the country and needless to mention the number of lives that would be saved.

Recently, telemedicine has gained a lot of importance and is practically being implemented in countries like France, Spain. But, till now Telemedicine is done in a ‘store and forward’ process where the reports of the patient are sent to doctors who are far away and they advise medication based on the reports. But with the development of Remote Patient



Visi Architecture

Monitoring, telemedicine will take a step forward. With advanced mobile technology, a doctor can monitor the status of a patient, get real-time reports on his/her medical status like Heart Rate, Blood Pressure. Even internal-body examinations can be done with the help of pill-shaped micro cameras that can travel inside the digestive system and can send images which helps in better diagnosis. one such system is ViSi Mobile that provides a complete solution to remote patient monitoring and is developed by Sotera Wireless.

The next step to remote patient controlling is “remote patient coaching”. Aoyama Morikawa Lab of University of Tokyo is presently developing a wearable computing outfit called “e-caching”, which provides coaching to the patients using them. The outfit first runs different tests on the body externally and takes reading and accordingly generates real-time coaching messages.

Embedded systems are also used to create artificial human organs. Even though the thought of putting something artificial inside your body might seem a little scary, artificial organs have a lot of applications. Apart from the fact that they can be replaced if any body-organ is malfunctioned, these artificial organs can be used to test drugs and medicine, which is presently done on animal organs.

A lot of organisations are into research and development of artificial organs; a San-Diego based company already has a working “mini-liver” which almost functions like the real one.

Smart architecture

With highly context aware and connected systems, Smart Homes have

become a reality. These homes have electronic circuits and sensors embedded into their walls and floors and everywhere else and they allow you to operate your entire house from just a one point contact such as a smartphone/tablet app. These days many companies are working on assistive homes that can help elderly and disabled people lead a quality life without the help of institutional care. These homes are equipped with a set of wireless systems connected with a central context aware systems for monitoring daily schedules of the elderly people and using machine learning algorithms to learn and recognise behavioral patterns. This enables such systems to detect abnormal activities and react to emergencies relating to medical conditions.

Offices are also using the Embedded systems technologies. These days many organisations have Smart Offices that have floors with load sensing systems. These load sensing systems can detect any unusual load and notify the authorities.

The concept of context aware devices is also used to design Green or Zero Net Energy Buildings (ZNEB). These buildings use intelligent energy management systems using a set of sensors and actuators such as heat sensors, light intensity sensors and magnetic door sensors. These systems automatically lower down your energy usages by detecting the context information such as whether a device is being used or not, whether you are in the room or not etc. One such model is being designed by General Electric.

Personal assistants

Future embedded systems will see a wide range of applications in the area of personal assistance devices. Google already is working on GLASS, a glass shaped head mounted display that is just a hands-free ubiquitous computer that lets you interact itself using voice commands such as “Glass, Where am I?” and “Glass, Take a picture”. Such devices act as smart and intelligent personal assistants. EMIEW 2 is an office assistant robot developed by Hitachi. It acts as your workmate and follows you voice instructions such as “follow me”. Whenever it sees an object, it takes its picture and compares it with an online database of images. Swiss scientists at the Artificial Lab of the University of Zurich are working on the most advanced humanoid personal assistant or service robot Roboy. Roboy is funded by a Kickstarter campaign and is a hugely anticipated robot.

Accessibility applications

Futuristic embedded systems facilitate numerous accessibility applications for navigating terrains that are rough and difficult to access. Scientists are working on swarm and insect size bots that can navigate vast areas and collect information quickly. At MIT media lab, researchers are working on a flying robot that is trained to track undersea objects such as whales and other marine organisms. Such robots can also be used to clean up toxic waste in chemical industries where it is harmful for human labor. Boston Dynamics created a robot called RHex, a six-legged robot with high mobility and ability to climb rock fields, sand, stairs, water, grass, and other difficult terrains etc. It has IR cameras that enable the operator to control it from far distances.

Smart retail

Today smart phones allow consumers to be connected to the brand and communicate with it in variety of ways. Today it has become imperative for consumer brands to understand the importance of Internet of Things concept and context awareness and apply them in business. With social media applications such as twitter and facebook, consumers stay connected to Brands all the time. Huggies launched an app called TweetPee that helps parents keep track of stock of diapers for their child and order online when needed. Using location meta -data, many brands are working on dynamic pricing of their products. With context aware systems, Brands can create competitive advantages that no other system can give them. There are certain billboards in Japan which instantly alter their displayed messages based on the profile of the consumer set looking at the billboard. This is made possible through image and video processing. Companies such as Tesco in South Korea are working on systems that will define the future of retail shopping. They created virtual stores with products integrated with QR(Quick Response) codes in subways where the busy Korean consumers could just use their smart phones to scan the product QR code and the product will be delivered to their doorstep.

Security and defense

Since long, one major application and research area for embedded systems has been security and defense, and even in future some of the most promising developments will be seen in this area. On simple application, “Capture Resistant Environment” is being developed at UbiComp lab in University of Washington.

This prevents unauthorised photography/videography by using projectors and cameras that make any such photo/video blurred. This enhances the security of areas which don't allow photography. Complex embedded systems are used to develop high-end reliable devices for the use of military and defense department. US Marine Corps use a robotic system called Gladiator that can perform functions ranging from surveillance to assault and minimises the risk of marines in threat situations. Today governments are shifting from drones to small sized bots that can easily navigate. Recently, UK Ministry announced Black Hornet , world's first "nano-sized" surveillance system which is a toy sized helicopter that can fly unmanned over an area for about 30 minutes on battery charge. US army is also working on the idea of combining a number of robot snakes to form a tentacle system that has the ability to operate like a human hand and can be used for life-threatening tasks such as Bomb Disposal.

Careers in embedded systems

After 10-20 years, Embedded systems are going to be everywhere - On every floor, every wall, your coffee mug, public transports, cars, homes, offices, aeroplanes. Looking at the numerous possible applications it has and the awesome projects that are already in pipeline, it is safe to say that Embedded Systems will be one of the most sought after fields of study and hence would require a lot of talented minds.

And it is also important to note that as an Embedded systems Engineer, one would be dealing with some of the most advanced and complicated technologies while working on systems that would revolutionise the way we look and feel this world. So, choosing this field as a career option now would be a smart option to people who always were interested in Electronics and Electrical Engineering.

Study and pre-requisites

Embedded systems is more like an amalgamation of almost all cool tech of today - Software Programming, Digital Electronics, Mobile Computing, Wearable Computing, Augmented Reality and hence would require a base knowledge of all the above mentioned fields, especially Digital Electronics.

Top Universities around the globe including MIT, Carnegie Mellon, Stanford, Cornell, University of Tokyo, University of Cambridge provide both Undergraduate and Postgraduate courses on Embedded systems. In India, major technological institutions like IITs, NITs, IIITs provide Post-Graduate courses on Embedded systems with a pre-requisite knowledge/

degree in Digital Electronics/ Electrical Engineering. These Institutions provide top-notch facilities, research laboratories and have a lot of ongoing projects which can give a deep insight into the study of this field.

A lot of Institutions also provide short-term Certification courses in Embedded systems such as Sastra University, University of pune, CDAC, IGNOU and a lot of training Institutions like Vector, Miracle Technologies and Thinklabs. But, these obviously lack the research facilities and environment as they are short term courses/certifications.

Specialisations

Even though Embedded Systems itself is a specialisation in Electronics/ Electrical Engineering, you can also work on your specific area of interest. Some of the options include Robotics, Automobiles, Mobile Technology, Wearable Computing, Augmented Reality. Although, there are no specialisation courses (like Embedded Systems in Robotics) in the above mentioned fields, one can do research/self projects in these fields and quote it as their Area of interest and apply in the respective organisations.

Career paths

Unlike the IT industry, the entry barriers in this field is quite high, due to the high level of expertise and experience required. Many of the current Embedded systems Engineers in India confess that the first major practical learning they had was in their first job. So, it is important that you end up in an organisation that has a work environment where you have the Opportunity to work on cutting edge technology. For this, you need to have an open mind to learn and be very passionate about technology and innovation. It is also needless to say that one has to be equipped with the right knowledge of both hardware and software, which is not that easy taking into account the variety of technology platforms in use. Some good research/self projects while graduation might help students secure a better job opportunity.

Embedded systems is used in almost every Industry today. Organisations varying from automobile manufacturing to mobile manufacturing need Embedded systems Engineers and Designers and it is hence fairly easy to switch between various companies and even industries.

A fresh college graduate has options of generally ending up as an Embedded Systems Engineer with an average salary of Rs. 4,00,000 - Rs. 8,00,000 and has a similar growth curve as the Engineers in IT Industry

– Engineers to Senior Engineers to Designers to Project Leads.

Companies that majorly recruit Embedded systems Engineers are Samsung, LG, HCL, IBM, National Instruments, Texas Instruments.

These companies also facilitate cutting edge research



MIT Robot Lab

labs. Many technology consulting firms like Persistent technologies also recruit Embedded systems Engineers at entry and senior levels.

So if you are planning to pursue Embedded systems as your career, get geared up to be part of a team that is working on a cool pocket robot or a complex mars rover. The field has endless opportunities in terms of growth and research and as the parallel technologies such as wireless communication and Artificial Intelligence keep growing, Embedded Systems will witness a complete overhaul in coming years.



All this and more in the
world of Technology

**VISIT
NOW**



www.thinkdigit.com

www.facebook.com

Join 1 lakh+ members of the Digit community



http://www.facebook.com/thinkdigit

facebook

digit
thinkdigit.com

Digit 1.5k likes
Product

Your favourite magazine on your social network. Interact with thousands of fellow Digit readers.

- Wall
- Info
- Photos
- Current News
- Schedule
- On Us, Off
- Connect It
- Profile
- More

http://www.facebook.com/IThinkGadgets

facebook

I Think Gadgets 1.5k likes
Community Center

An active community for those of you who love mobiles, laptops, cameras and other gadgets. Learn and share more on technology.

- Wall
- Info
- Photos
- Videos
- Events
- Questions

http://www.facebook.com/GladtobeAProgrammer

facebook

Glad to be a Programmer 1.5k likes
Programmer

If you enjoy writing code, this community is for you. Be a part and find your way through development.

- Wall
- Info
- Photos
- Videos
- Pages
- Events

http://www.facebook.com/devworx.in

facebook

devworx 1.5k likes
Software Developer, India

devworx, a niche community for software developers in India, is supported by 9.9 Media, publishers of Digit

- Wall
- Info
- Friend Activity (2)
- Questions
- Tips
- Photos
- Wallpapers
- More