SMART TRAFFIC MANAGEMENT SYSTEM

A PROJECT REPORT

submitted by

Sai Khushee
230701276
Sentamizh
230701301
Shankaranarayanan
230701306

in partial fulfillment for the award of the

degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

RAJALAKSHMI ENGINEERING COLLEGE

BONAFIDE CERTIFICATE

Certified that this project **"Smart Traffic Management System"** is the bonafide work of *Sai Khushee, Shankaranarayanan, Sentamizh* who carried out the project work under my supervision.

Faculty Incharge

Faculty name: Ponmani S

Designation:

Department of Computer Science and Engineering, Rajalakshmi Engineering College, Chennai

This mini project report is submitted for the viva vice examination to be held on __________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

OBJECTIVE:

The objective of this project is to design and implement an intelligent traffic light management system that dynamically adjusts the lane opening time based on real-time traffic density using IR sensors. Each lane is equipped with two IR sensors that count the number of vehicles, and if both sensors detect vehicle presence, the lane's green signal duration is increased. Additionally, an RFID-based mechanism is integrated to prioritize emergency vehicles such as ambulances, fire engines, and police cars. When an emergency vehicle with an RFID tag approaches, the system identifies it and automatically opens the lane to ensure uninterrupted passage.

ABSTRACT:

With increasing vehicle numbers and urban congestion, managing traffic effectively has become a major challenge. Traditional traffic light systems operate on fixed timers and do not adapt to varying traffic densities, leading to inefficiencies and longer wait times. This project proposes a smart traffic control system using IR sensors and RFID technology to optimize traffic flow and prioritize emergency vehicles.

Each lane is equipped with two IR sensors to monitor traffic density. When both sensors detect vehicle presence, it indicates a higher vehicle count, and the green light duration is extended for that lane. This ensures that heavily congested lanes receive more time, thereby reducing traffic buildup. Moreover, the system includes an RFID reader to detect emergency vehicles. When an emergency vehicle is near and its RFID tag is read, the traffic light instantly turns green for that lane, allowing quick and safe passage.

This automated system not only reduces congestion and wait times but also ensures emergency services are not delayed. By integrating

low-cost sensors and microcontrollers, the proposed solution is affordable and scalable for smart city implementations.

INTRODUCTION:

Urban areas are facing increasing challenges in managing traffic due to the rapid rise in vehicle usage. Traditional traffic lights operate on preset timers that fail to adapt to real-time traffic conditions, leading to inefficient traffic flow and frustration among commuters. Delays caused by traffic congestion also pose serious problems for emergency services that require swift navigation through busy streets.

A smart traffic management system is the need of the hour to reduce travel time, improve fuel efficiency, and enhance road safety. Modern technology, including embedded systems, IR sensors, and RFID, provides an opportunity to revolutionize how traffic is controlled. By using IR sensors placed at strategic points in each lane, we can measure vehicle presence and estimate traffic density. This data is then used to control the duration of the traffic signals dynamically.

Emergency response time is crucial in saving lives. Emergency vehicles often get stuck in traffic, despite using sirens. By integrating an RFID reader, emergency vehicles with an RFID tag can be detected, prompting the system to automatically grant them passage. This intelligent solution not only saves lives but also increases the efficiency of the entire traffic ecosystem.

The integration of IR sensors for density monitoring and RFID for emergency access provides a holistic approach to modern traffic management, making it a step forward in building smart cities.

EXISTING SYSTEM :

In the existing traffic management systems, traffic lights operate on a fixed timing mechanism. The signal changes are based on predetermined time intervals regardless of the actual number of vehicles waiting at each lane. Traffic police may manually control the lights during peak hours or special situations, but this process is inefficient, labor-intensive, and prone to human error.

Additionally, emergency vehicles often struggle to navigate through traffic as there is no automated mechanism to detect their presence and adjust traffic signals accordingly. Delays in emergency services can result in severe consequences, especially in medical emergencies, fire accidents, and police interventions.

PROPOSED SYSTEM:

The proposed system utilizes two IR sensors per lane to estimate vehicle density. If both IR sensors in a lane detect vehicles, it indicates high traffic, prompting the microcontroller to increase the green light duration for that specific lane. This allows more vehicles to pass through and reduces congestion effectively.

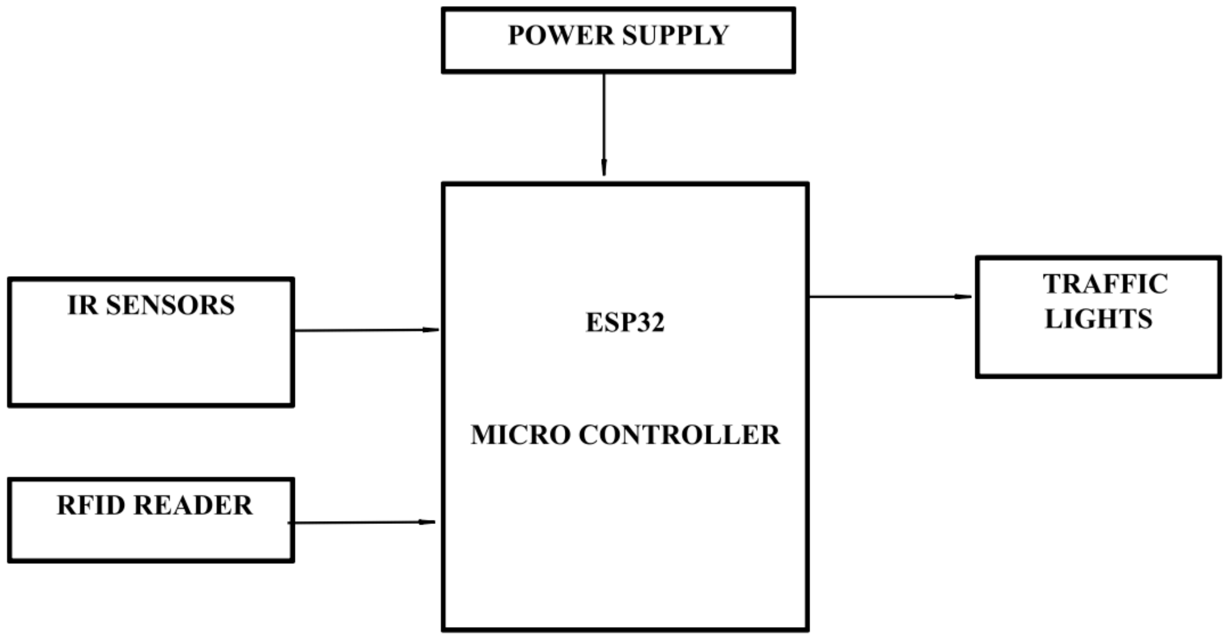
To handle emergency vehicles, each is fitted with an RFID tag. An RFID reader installed near the intersection detects the tag as the vehicle approaches. Upon recognizing a valid emergency RFID, the traffic light for that lane immediately turns green, regardless of its previous state. This ensures emergency vehicles get the right of way without delay.

The system is controlled using a microcontroller (like ESP32 or Arduino), and the logic is coded to manage the traffic lights based on sensor input. The entire process is automatic and requires minimal human intervention, leading to efficient and intelligent traffic control.

ADVANTAGES:

- **Real-time traffic detection** using IR sensors enhances efficiency.
- **Dynamic signal timing** improves vehicle flow and reduces waiting time.
- **Priority access for emergency vehicles** through RFID ensures public safety.
- **Minimizes human intervention**, reducing the chance of error.
- **Energy-efficient and cost-effective** system implementation.
- **Scalable and adaptable** for future smart city applications.
- **Reduces fuel consumption** and air pollution due to less idle time.
- **Improves emergency response time**, potentially saving lives.

Block Diagram:



HARDWARE REQUIREMENTS

- ESP32 MICROCONTROLLER
- POWER SUPPLY
- IR SENSOR
- RFID READER
- TRAFFIC LIGHT

SOFTWARE REQUIREMENTS

- EMBEDDED C
- ARDUINO IDE

HARDWARE DESCRIPTION

POWER SUPPLY ADAPTER

GENERAL DESCRIPTION

An adapter is a device that converts attributes of one electrical device or system to those of an otherwise incompatible device or system. Some modify power or signal attributes, while others merely adapt the physical form of one electrical connector to another. In a computer, an adapter is often built into a card that can be inserted into a slot on the computer's motherboard. The card adapts information that is exchanged between the computer's microprocessor and the devices that the card supports.

PRODUCT DESCRIPTION

An electric power adapter may enable connection of a power plug, sometimes called, used in one region to a AC power socket used in another, by offering connections for the disparate contact arrangements, while not changing the voltage. An AC adapter, also called a "recharger", is a small power supply that changes household electric current from distribution voltage) to low voltage DC suitable for consumer electronics. Some modify power or signal attributes, while others merely adapt the physical form of one electrical connector to another. For computers and related items, one kind of serial port adapter enables connections between 25-contact and nine-contact connectors, but does not affect electrical power- and signalling-related attributes.

FEATURES:

· Output current:1A

· Supply voltage: 220-230VAC

· Output voltage: 12VDC

· Reduced costs

· Increased value across front-office and back-office functions

· Access to current, accurate, and consistent data

· It generates adapter metadata as WSDL files with J2CA extension.

APPLICATIONS:

· Back-end systems which need to send purchase order data to oracle applications send it to the integration service via integration server client.

· SMPS applications.

MICROCONTROLLER

ESP32

INTRODUCTION

Arduino is a great platform for beginners into the World of Microcontrollers and Embedded Systems. With a lot of cheap sensors and modules, you can make several projects either as a hobby or even commercial.

As technology advanced, new project ideas and implementations came into play and one particular concept is the Internet of Things or IoT. It is a connected platform, where several "things" or devices are connected over the internet for exchange of information.

In the DIY community, the IOT projects are mainly focused on Home Automation and Smart Home applications but commercial and industrial IoT projects have far more complex implementations like Machine Learning, Artificial Intelligence, Wireless Sensor Networks etc.

The important thing in this brief intro is whether it is a small DIY project by a hobbyist or a complex industrial project, any IoT project must have connectivity to the Internet. This is where the likes of ESP8266 and ESP32 come into picture.

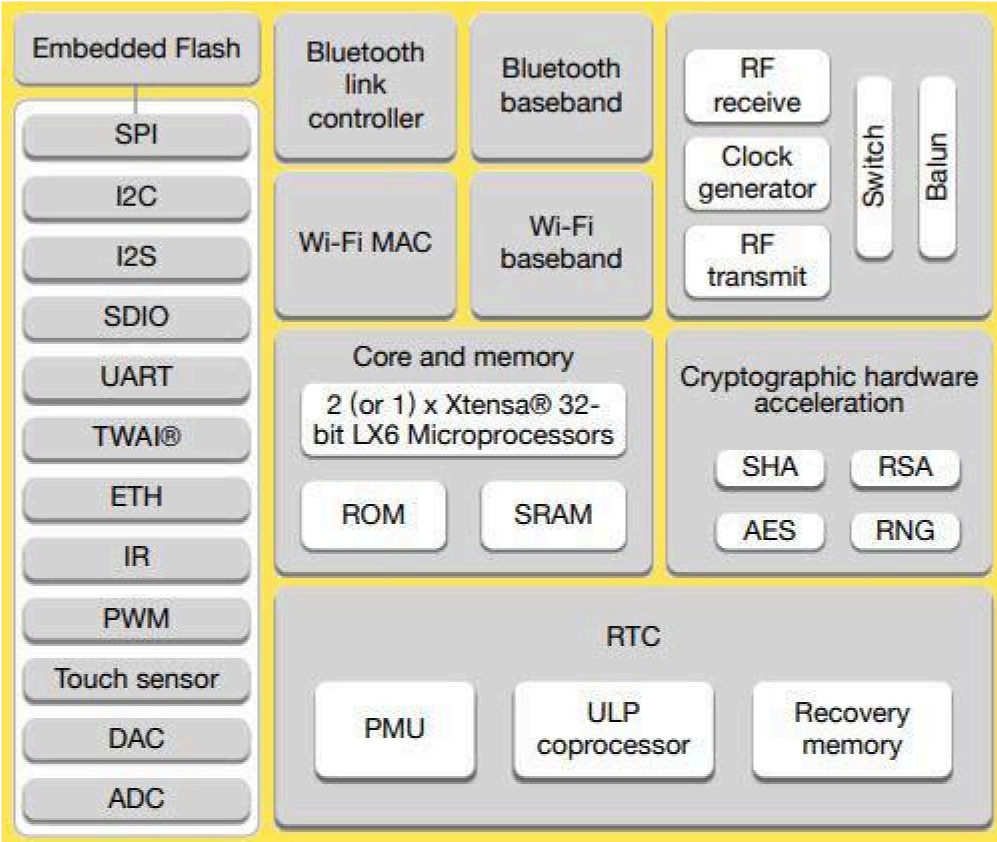
If you want to add Wi-Fi connectivity to your projects, then ESP8266 is a great option. But if you want to build a complete system with Wi-Fi connectivity, Bluetooth connectivity, high resolution ADCs, DAC, Serial Connectivity and many other features, then ESP32 is the ultimate choice.

What is ESP32?

ESP32 is a low-cost System on Chip (SoC) Microcontroller from Espressif Systems, the developers of the ESP8266 SoC. It is a successor to ESP8266 SoC and comes in both single-core and dual-core variations of Tensilica's 32-bit Xtensa LX6 Microprocessor with integrated Wi-Fi and Bluetooth.

The good thing about ESP32, like ESP8266 is its integrated RF components like Power Amplifier, Low-Noise Receive Amplifier, Antenna Switch, Filters and RF Balun. This makes designing hardware around ESP32 very easy as you require very few external components.

Another important thing to know about ESP32 is that it is manufactured using TSMC’s ultra-low-power 40 nm technology. So, designing battery operated applications like wearables, audio equipment, baby monitors, smart watches, etc., using ESP32 should be very easy.



GENERAL DESCRIPTION:

ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. The ESP32 series employs either a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core variations, Xtensa LX7 dual-core microprocessor or a single-core RISC-V microprocessor and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. ESP32 is created and developed by Espressif Systems, a Shanghai-based Chinese company, and is manufactured by TSMC using their 40 nm process.

Specifications of ESP32

ESP32 has a lot more features than ESP8266 and it is difficult to include all the specifications in this Getting Started with ESP32 guide. So, I made a list of some of the important specifications of ESP32 here. But for a complete set of specifications, I strongly suggest you refer to the Datasheet.

- Single or Dual-Core 32-bit LX6 Microprocessor with clock frequency up to 240 MHz.
- 520 KB of SRAM, 448 KB of ROM and 16 KB of RTC SRAM.
- Supports 802.11 b/g/n Wi-Fi connectivity with speeds up to 150 Mbps.
- Support for both Classic Bluetooth v4.2 and BLE specifications.

- 34 Programmable GPIOs.
- Up to 18 channels of 12-bit SAR ADC and 2 channels of 8-bit DAC
- Serial Connectivity include 4 x SPI, 2 x I^2C, 2 x I^2S, 3 x UART.
- Ethernet MAC for physical LAN Communication (requires external PHY).
- 1 Host controller for SD/SDIO/MMC and 1 Slave controller for SDIO/SPI.
- Motor PWM and up to 16-channels of LED PWM.
- Secure Boot and Flash Encryption.
- Cryptographic Hardware Acceleration for AES, Hash (SHA-2), RSA, ECC and RNG.

IR SENSOR

GENERAL DESCRIPTION

IR LED emits infrared radiation. This radiation illuminates the surface in front of the LED. Depending on the reflectivity of the surface, the amount of light reflected varies. This reflected light is made on a reverse biased IR sensor. The amount of electron-hole pairs generated depends on intensity of incident IR radiation. Thus as intensity of incident ray varies, voltage across resistor will vary accordingly.

PRODUCT DESCRIPTION

An infrared sensor is an electronic device that emits in order to sense some aspects of the surroundings. An IR sensor can measure the heat of an object as well as detect the motion. Usually in the infrared spectrum, all the objects radiate some form of thermal radiations. These types of radiation are invisible to our eyes, and can be detected by an infrared sensor. The emitter is simply an IR LED (Light Emitting Diode) and the detector is simply an IR photodiode which is sensitive to IR light of the same wavelength as that emitted by the IR LED. When IR light falls on the photodiode, The resistances and these output voltages change in proportion to the magnitude of the IR light received.

FEATURES

· Operating voltage:5VDC

· Output voltage: 0 or 5VDC

· Easy to assemble and use

· Onboard detection indication

· Effective distance range of 2cm

APPLICATIONS

· Augmentative communication devices

· Car locking systems

· Computers

· Signage

· Telephones

RFID READER

GENERAL DESCRIPTION

A Radio Frequency Identification Reader (RFID reader) is a device used to gather information from an RFID tag, which is used to track individual objects. Radio Frequency waves are used to transfer data from the tag to a reader. The RFID tag must be within the range of an RFID reader, in order to be read. RFID technology allows several items to be quickly scanned and enables fast identification of a particular product, even when it is surrounded by several other items.

PRODUCT DESCRIPTION

Radio frequency identification (RFID) is one method for Automatic Identification and Data Capture (AIDC). RFID tags are used in many industries. An RFID system consists of three components: an antenna and transceiver and a transponder. The antenna uses radio frequency waves to transmit a signal that activates the transponder. When activated, the tag transmits data back to the antenna. An RFID reader's function is to interrogate RFID tags. The means of interrogation is wireless and because the distance is relatively short; line of sight between the reader and tags is not necessary. A reader contains an RF module, which acts as both a transmitter and receiver of radio frequency signals. The transmitter consists of an oscillator to create the carrier frequency; a modulator to impinge data commands upon this carrier signal and an amplifier to boost the signal enough to awaken the tag. The receiver has a demodulator to extract the returned data and also contains an amplifier to strengthen the signal for processing. A microprocessor forms the control unit, which employs an operating system and memory to filter and store the data. The data is now ready to be sent to the network.

RFID Reader

FEATURES

- Supply voltage: 12v DC

- Output: UART and TTL

- In-built buzzer indicator
- Signal LED is placed

APPLICATIONS

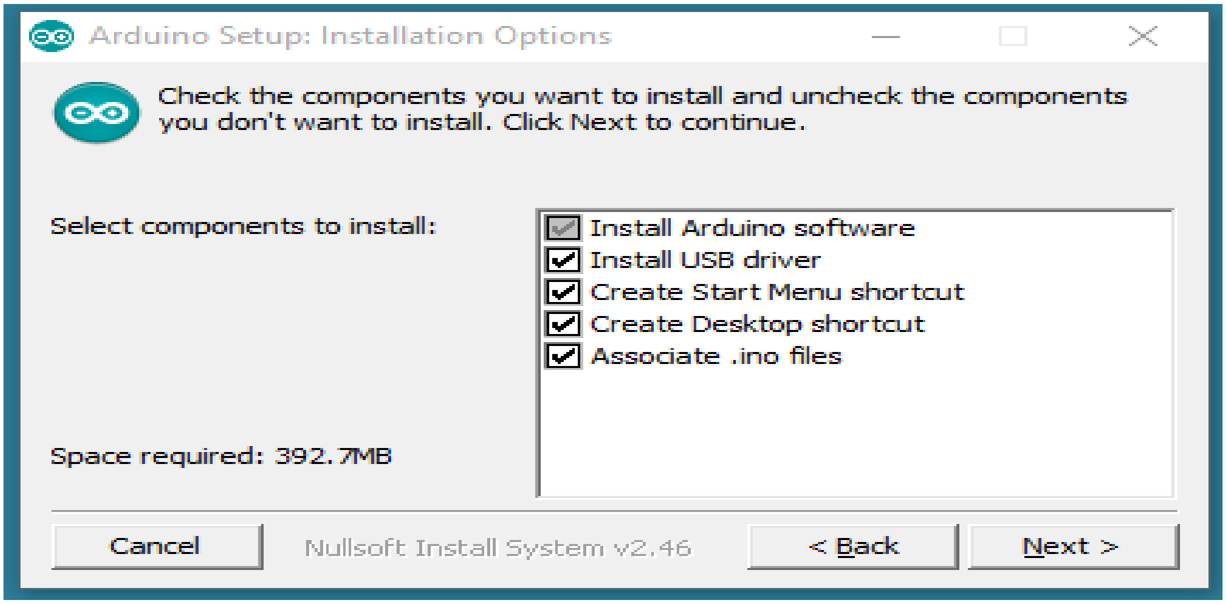
- Passports
- Toll booth passes
- Hospitals
- Libraries

SOFTWARE DESCRIPTION

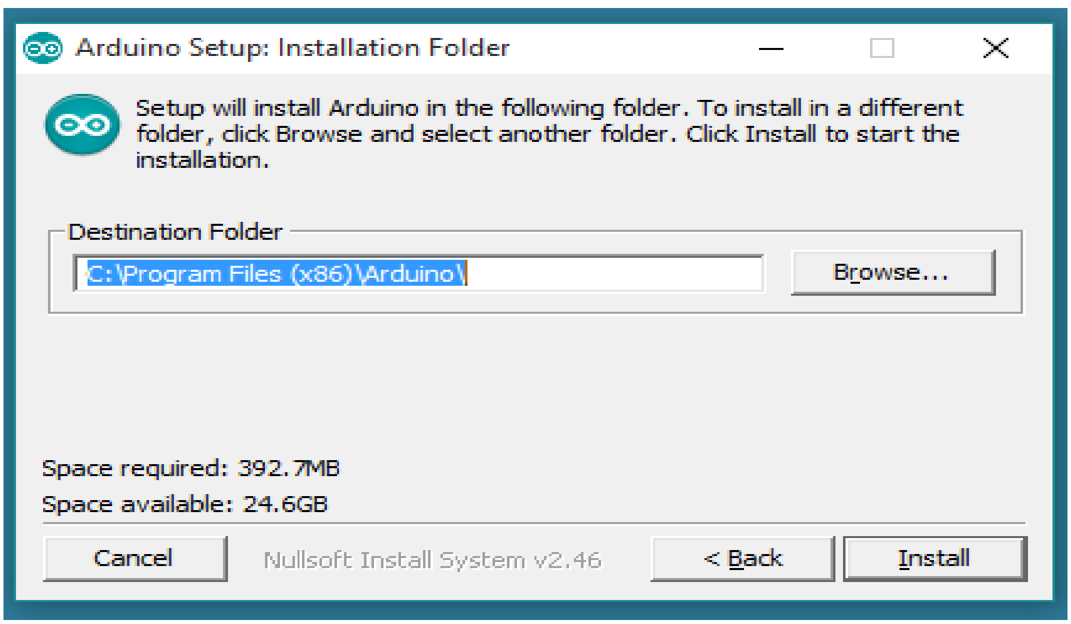
Arduino Software (IDE)

Get the latest version from the download page. You can choose between the Installer (.exe) and the Zip packages. We suggest you use the first one that installs directly everything you need to use the Arduino Software (IDE), including the drivers. With the Zip package you need to install the drivers manually. The Zip file is also useful if you want to create a portable installation.

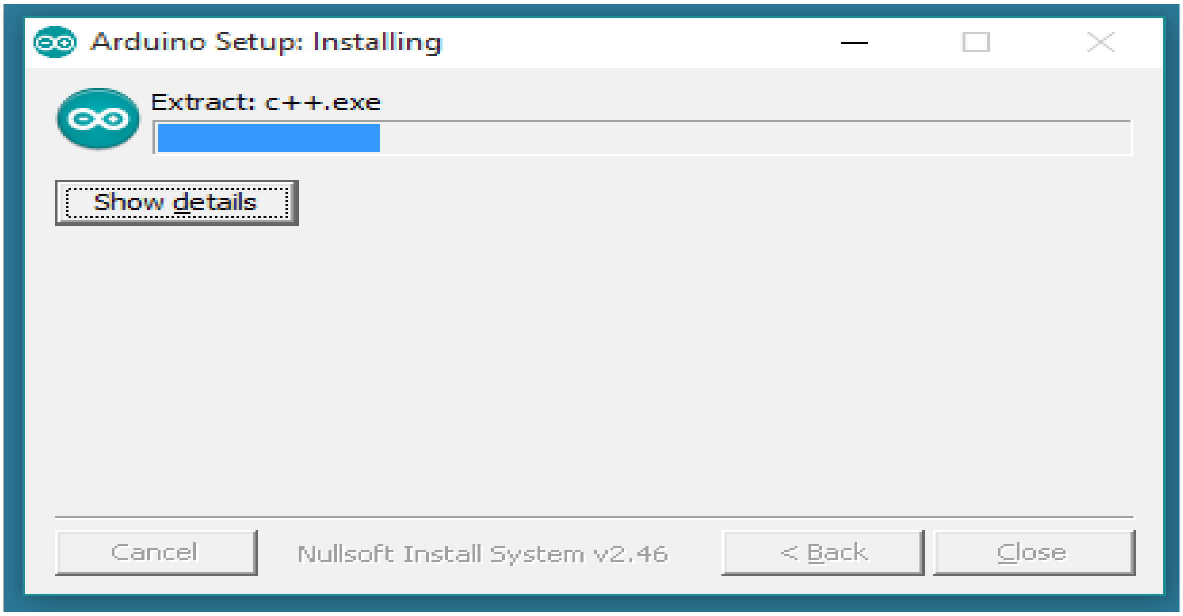
When the download finishes, proceed with the installation and please allow the driver installation process when you get a warning from the operating system.



Choose the components to install



Choose the installation directory (we suggest to keep the default one)



The process will extract and install all the required files to execute properly the Arduino Software (IDE)

Arduino Bootloader Issue

The current boot loader burned onto the Arduino UNO is not compatible with ROBOTC. In its current form, you will be able to download the ROBOTC Firmware to the ArduinoUNO, but you will not able to download any user programs.

The reason for this is because there is a bug in the Arduino UNO firmware that does not allow flash write commands to start at anywhere but the beginning of flash memory (0x000000). See the bottom of this page for more technical details.

Because ROBOTC is not able to burn a new bootloader as of today, you will need to use the Arduino's Open Source language with a modified bootloader file to re-burn your bootloader on your Arduino UNO boards. The enhanced bootloader is backwards compatible with the original one.

That means you'll still be able to program it through the Arduino programming environment as before, in addition to ROBOTC for Arduino.

Hardware Needed

To burn a new version of the Arduino bootloader to your UNO, you'll need an AVR ISP Compatible downloader.

Using an AVR ISP (In System Programmer)

- Your Arduino UNO (to program)
- An AVR Programmer such as the AVR Pocket Programmer
- An AVR Programming Cable (the pocket programmer comes with one)

If you have extra Arduino boards, but no ISP programmer, SparkFun.com has a cool tutorial on how to flash a bootloader using an Arduino as an ISP.

Using another Arduino as an ISP

- Your Arduino UNO (to program)
- A Working Arduino (doesn't matter what kind)
- Some Male-to-Male Jumper Cables.

Software Needed

ROBOTC is not currently able to burn a bootloader onto an Arduino board, so you'll need to download a copy of the latest version of the Arduino Open-Source programming language.

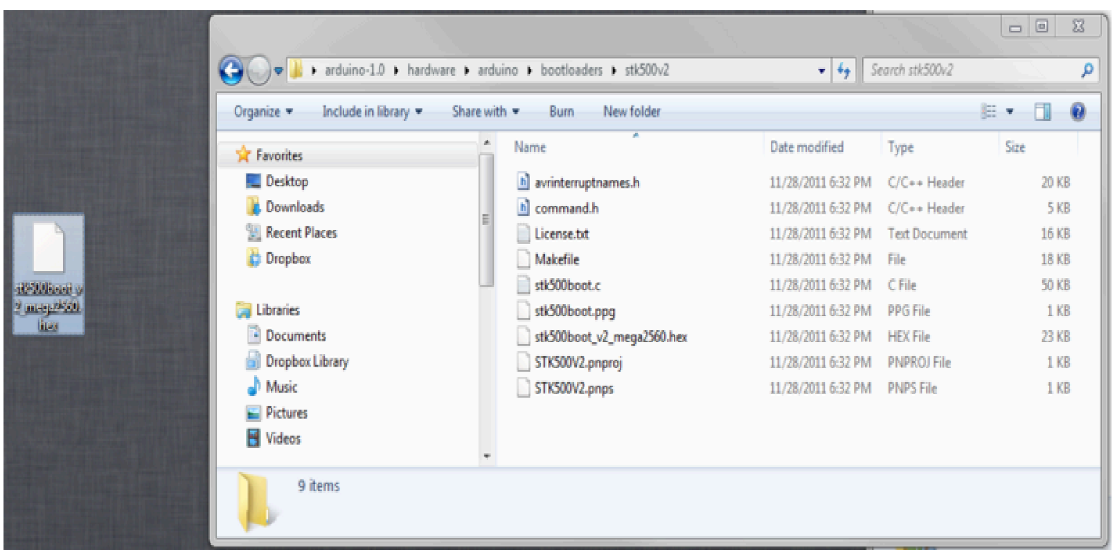
- Arduino Official Programming Language - Download Page

In addition, you'll need the ROBOTC modified bootloader. You can download that here:

- ROBOTC Modified UNO Bootloader - Modified Bootloader

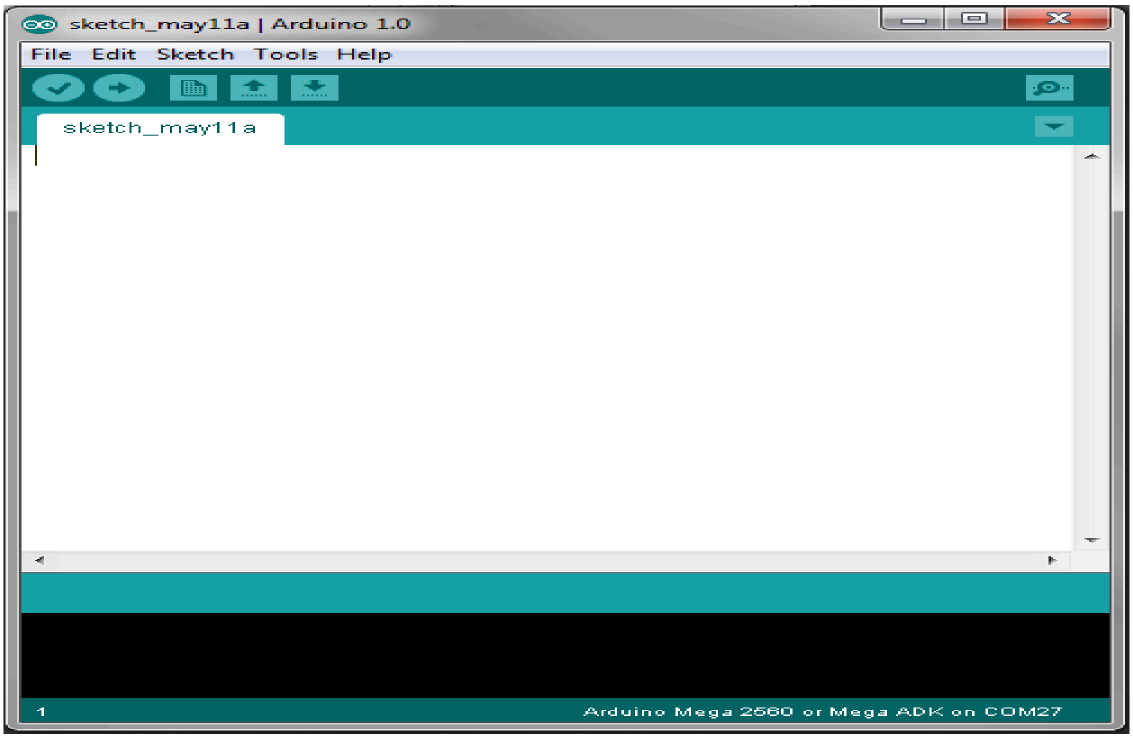
Bootloader Download Instructions

- Download the Arduino Open Source Software and a copy of the Modified Bootloader File
- Copy the Modified Bootloader File into the /Arduino-1.0/hardware/arduino/bootloaders/stk500v2/ and overwrite the existing bootloader.



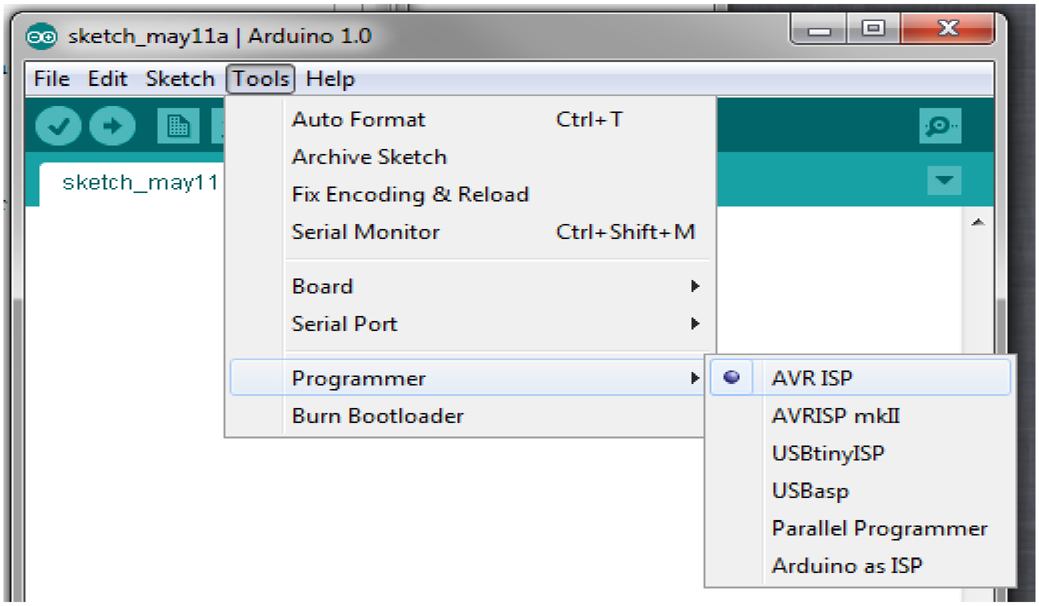
- Power up your Arduino UNO (either via USB or external power)

- Plug in your AVR ISP Programmer to your computer (make sure you have any required drivers installed)
- Connect your AVR ISP Programmer into your Arduino UNO Board via the ISP Header (the 2x3 header pins right above the Arduino Logo)
- Launch the Arduino Open Source Software

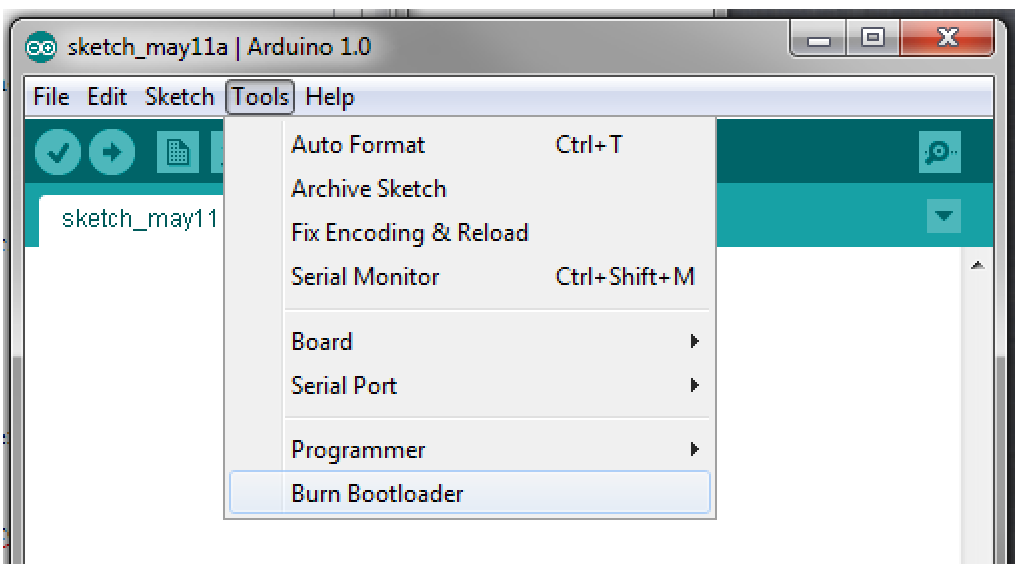


- Change your settings in the Arduino Software to look for an Arduino UNO

- Change your settings in the Arduino Software to select your ISP Programmer Type (Check your programmer's documentation for the exact model)



- Select the "Burn Bootloader" option under the "Tools" menu. The modified bootloader will now be sent to your Arduino. This typically take a minute or so.



- You should be all set to download ROBOTC firmware and start using your Arduino UNO with ROBOTC.

Technical Details

The Arduino Bootloader sets the "erase Address" to zero every time the boot loader is called. ROBOTC called the "Load Address" command to set the address in which we want to write/verify when downloading a program.

When writing a page of memory to the arduino, the Arduino bootloader will erase the existing page and write a whole new page. In the scenario of downloading firmware, everything is great because the Erase Address and the Loaded Address both start at zero.

In the scenario of writing a user program, we start writing at memory location 0x7000, but the Bootloader erases information starting at location zero because the "Load Address" command doesn't update where to erase.

Our modification is to set both the Load Address and the Erase Address so the activity of writing a user program doesn't cause the firmware to be accidentally erased.

Summary

Microcontroller	Arduino UNO
Operating Voltage	5V Input Voltage (recommended)
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 14 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40mA

DC Current for3.3VPin	50mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8KB
EEPROM	4KB
Lock Speed	16MHz

The Arduino UNO can be powered via the USB connection or with an external power supply. The power source is selected automatically. External (non-USB) power can come either from an AC-to-DC adapter (wall-wart) or battery. The adapter can be connected by plugging a 2.1mm center-positive plug into the board's power jack. Leads from a battery can be inserted in the Gnd and V_{in} pin headers of the POWER connector.
The board can operate on an external supply of 6 to 20 volts. If supplied with less than 7V, however, the 5V pin may supply less than five volts and the board may be unstable. If using more than 12V, the voltage regulator may overheat and damage the board. The recommended range is 7 to 12 volts.

They differ from all preceding boards in that it does not use the FTDI USB-to-serial driver chip. Instead, it features the program as a USB-to-serial converter.

The power pins are as follows:

- **VIN.** The input voltage to the Arduino board when it's using an external power source (as opposed to 5 volts from the USB connection or other regulated power source). You can supply voltage through this pin, or, if supplying voltage via the power jack, access it through this pin.

- **5V.** The regulated power supply used to power the microcontroller and other components on the board. This can come either from VIN via a non-board regulator, or be supplied by USB or another regulated 5V supply.
- **3V3.**A3.3voltsupplygeneratedbytheon-boardregulator.Maximumcurrentdrawis50mA.
- **GND.** Ground pins.

The ATMEGA has 256 KB of flash memory for storing code (of which 8 KB is used for the bootloader), 8 KB of SRAM and 4 KB of EEPROM (which can be read and written with the EEPROM library).

Each of the 54 digital pins on the Mega can be used as an input or output, using pinMode(), digitalWrite(), and digitalRead() functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50k Ohms. In addition, some pins have specialized functions:

- **Serial: 0 (RX) and 1 (TX); Serial 1: 19 (RX) and 18 (TX); Serial 2: 17 (RX) and 16 (TX); Serial 3: 15 (RX) and 14 (TX).** Used to receive (RX) and transmit (TX) TTL serial data. Pins 0 and 1 are also connected to the corresponding pins of the ATMEGA USB-to-TTL Serial chip.

- **External Interrupts: 2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), and 21 (interrupt 2).** These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a changing value. See the attach Interrupt() function for details.

- **PWM: 0to13.** Provide 8-bit PWM output with the analogWrite() function.
- **SPI: 50 (MISO), 51 (MOSI), 52 (SCK), 53 (SS).** These pins support SPI communication, which, although provided by the underlying hardware, is not currently included in the Arduino language. The SPI pins are also broken out on the ICSP header, which is physically compatible with the Duemilanove and Diecimila.
- **LED: 13.** There is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.
- **I^2C: 20 (SDA) and 21 (SCL).** Support I^2C (TWI) communication using the Wire library (documentation on the Wiring website). Note that these pins are not in the same location as the I^2C pins on the Duemilanove.

The Arduino UNO has 16 analog inputs, each of which provide 10 bits of resolution (i.e. 1024 different values). By default they measure from ground to 5 volts, though is it possible to change the upper end of their range using the AREF pin and analog Reference() function.

There are a couple of other pins on the board:

- **AREF.** Reference voltage for the analog inputs. Used with analog Reference().
- **eset.** Bring this line LOW to reset the microcontroller. Typically used to add a reset button to shields which block the one on the board.

Communication

The Arduino UNO has a number of facilities for communicating with a computer, another Arduino, or other microcontrollers. The Arduino UNO provides four hardware UARTs for TTL (5V) serial communication.

An ATMEGA on the board channels one of these over USB and provides a virtual comport to software on the computer (Windows machines will need a .inf file, but OSX and Linux machines will recognize the board as a COM port automatically. The Arduino software includes a serial monitor which allows simple textual data to be sent to and from the board. The RX and TX LEDs on the board will flash when data is being transmitted via the ATmega8U2 chip and USB connection to the computer (but not for serial communication on pins 0 and1).

A SoftwareSerial library allows for serial communication on any of the digital pins. The Arduino UNO also supports I2C (TWI) and SPI communication. The Arduino software includes a Wire library to simplify use of the I2C bus; see the documentation on the Wiring website for details. To use the SPI communication, please see the Arduino UNO datasheet.

Programming

The Arduino UNO can be programmed with the Arduino software (download). For details, see the reference and tutorials.

The Arduino UNO comes pre-burned with a bootloader that allows you to upload new code to it without the use of an external hardware programmer. It communicates using the original STK500 protocol (reference, C header files).

You can also bypass the bootloader and program the microcontroller through the ICSP (In-Circuit Serial Programming) header; see these instructions for details.

Automatic (Software) Reset

Rather than requiring a physical press of the reset button before an upload, the Arduino UNO is designed in a way that allows it to be reset by software running on a connected computer. One of the hardware flow control lines (DTR) of the ATmega8U2 is connected to the reset line of the Arduino UNO via a 100 nanofarad capacitor. When this line is asserted (taken low), the reset line drops long enough to reset the chip. The Arduino software uses this capability to allow you to upload code by simply pressing the upload button in the Arduino environment. This means that the bootloader can have a shorter timeout, as the lowering of DTR can be well-coordinated with the start of the upload.

This setup has other implications. When the Arduino UNO is connected to either a computer running Mac OS X or Linux, it resets each time a connection is made to it from software (via USB). For the following half-second or so, the bootloader is running on the UNO. While it is programmed to ignore malformed data (i.e. anything besides an upload of new code), it will intercept the first few bytes of data sent to the board after a connection is opened. If a sketch running on the board receives one-time configuration or other data when it first starts, make sure that the software with which it communicates waits a second after opening the connection and before sending this data.

The Mega contains a trace that can be cut to disable the auto-reset. The pads on either side of the trace can be soldered together to re-enable it. It's labeled "RESET-EN". You may also be able to disable the auto-reset by connecting a 110 ohm resistor from 5V to the reset line; see this forum thread for details.

USB Over current Protection

Physical Characteristics and Shield Compatibility

The Arduino UNO has a resettable poly fuse that protects your computer's USB ports from shorts and over current. Although most computers provide their own internal protection, the fuse provides an extra layer ofprotection.Ifmorethan500mAisappliedtotheUSBport, the fuse will automatically break the connection until the short or overload is removed.

The maximum length and width of the UNO PCB are 4 and 2.1 inches respectively, with the USB connector and power jack extending beyond the former dimension. Three screw holes allow the board to be attached to a surface or case. Note that the distance between digital pins 7 and 8 is 160 mil (0.16"), not an even multiple of the 100 mil spacing of the other pins.

The UNO is designed to be compatible with most shields designed for the Diecimila or Duemilanove. Digital pins 0 to 13 (and the adjacent AREF and GND pins), analog inputs 0 to 5, the power header, and ICSP header are all in equivalent locations. Further the main UART (serial port) is located on the same pins (0 and 1), as are external interrupts 0 and 1 (pins 2 and 3 respectively). SPI is available through the ICSP header on both the Mega and Duemilanove / Diecimila.

Please note that I^2C is not located on the same pins on the Mega (20and21) as the Duemilanove / Diecimila (analog inputs 4 and 5).

How to use Arduino

Arduino can sense the environment by receiving input from a variety of sensors and can affect its surroundings by controlling lights, motors, and other actuators. The microcontroller on the board is programmed using the Arduino programming language (based on Wiring) and the Arduino development environment (based on Processing). Arduino projects can be stand-alone or they can communicate with software on running on a computer (e.g. Flash, Processing, MaxMSP).

Arduino is a cross-platform program. You'll have to follow different instructions for your personal OS. Check on the Arduino site for the latest instructions. *http://arduino.cc/en/Guide/HomePage*
Once you have downloaded/unzipped the arduino IDE, you can plug the Arduino to your PC via USB cable.

EMBEDDED C

ABOUT EMBEDDED C

High-level language programming has long been in use for embedded-systems development. However, assembly programming still prevails, particularly for digital-signal processor (DSP) based systems. DSPs are often programmed in assembly language by programmers who know the processor architecture inside out. The key motivation for this practice is performance, despite the disadvantages of assembly programming when compared to high-level language programming.

If the video decoding takes 80 percent of the CPU-cycle budget instead of 90 percent, for instance, there are twice as many cycles available for audio processing. This coupling of performance to end-user features is characteristic of many of the real-time applications in which DSP processors are applied. DSPs have a highly specialized architecture to achieve the performance requirements for signal processing applications within the limits of cost and power consumption set for consumer applications. Unlike a conventional Load-Store (RISC) architecture, DSPs have a data path with memory-access units that directly feed into the arithmetic units. Address registers are taken out of the general-purpose register file and placed next to the memory units in a separate register file.

A further specialization of the data path is the coupling of multiplication and addition to form a single cycle Multiply-accumulate unit (MAC). It is combined with special-purpose accumulator registers, which are separate from the general-purpose registers. Data memory is segmented and placed close to the MAC to achieve the high bandwidths required to keep up with the streamlined data path. Limits are often placed on the extent of memory-addressing operations. The localization of resources in the data path saves many data movements that typically take place in a Load-Store architecture.

The most important, common arithmetic extension to DSP architectures is the handling of saturated fixed-point operations by the arithmetic unit. Fixed-point arithmetic can be implemented with little additional cost over integer arithmetic. Automatic saturation (or clipping) significantly reduces the number of control-flow instructions needed for checking overflow explicitly in the program.Changes in technological and economic requirements make it more expensive to continue programming DSPs in assembly. Staying with the mobile phone as an example, the signal-processing algorithms required become increasingly complex. Features such as stronger error correction and encryption must be added. Communication protocols become more sophisticated and require much more code to implement. In certain markets, multiple protocol stacks are implemented to be compatible with multiple service providers. In addition, backward compatibility with older protocols is needed to stay synchronized with provider networks that are in a slow process of upgrading.

Today, most embedded processors are offered with C compilers. Despite this, programming DSPs is still done in assembly for the signal processing parts or, at best, by using assembly-written libraries supplied by manufacturers. The key reason for this is that although the architecture is well matched to the requirements of the signal-processing application, there is no way to express the algorithms efficiently and in a natural way in Standard C. Saturated arithmetic.

For example, it is required in many algorithms and is supplied as a primitive in many DSPs. However, there is no such primitive in Standard C. To express saturated arithmetic in C requires comparisons, conditional statements, and correcting assignments. Instead of using a primitive, the operation is spread over a number of statements that are difficult to recognize as a single primitive by a compiler.

DESCRIPTION

Embedded C is designed to bridge the performance mismatch between Standard C and the embedded hardware and application architecture. It extends the C language with the primitives that are needed by signal-processing applications and that are commonly provided by DSP processors. The design of the support for fixed-point data types and named address spaces in Embedded C is based on DSP-C. DSP-C [1] is an industry-designed extension of C with which experience was gained since 1998 by various DSP manufacturers in their compilers. For the development of DSP-C by ACE (the company three of us work for), cooperation was sought with embedded-application designers and DSP manufacturers.

The Embedded C specification extends the C language to support freestanding embedded processors in exploiting the multiple address space functionality, user-defined named address spaces, and direct access to processor and I/O registers. These features are common for the small, embedded processors used in most consumer products. The features introduced by Embedded C are fixed-point and saturated arithmetic, segmented memory spaces, and hardware I/O addressing. The description we present here addresses the extensions from a language-design perspective, as opposed to the programmer or processor architecture perspective.

MULTIPLE ADDRESS SPACES

Embedded C supports the multiple address spaces found in most embedded systems. It provides a formal mechanism for C applications to directly access (or map onto) those individual processor instructions that are designed for optimal memory access. Named address spaces use a single, simple approach to grouping memory locations into functional groups to support MAC buffers in DSP applications, physical separate memory spaces, direct access to processor registers, and user-defined address spaces.

The Embedded C extension supports defining both the natural multiple address space built into a processor's architecture and the application-specific address space that can help define the solution to a problem.

Embedded C uses address space qualifiers to identify specific memory spaces in variable declarations. There are no predefined keywords for this, as the actual memory segmentation is left to the implementation. As an example, assume that **X** and **Y** are memory qualifiers. The definition:

```
X int a[25] ;
```

Means that **a** is an array of 25 integers, which is located in the **X** memory. Similarly (but less common):

```
X int * Y p ;
```

Means that the pointer **p** is stored in the **Y** memory. This pointer points to integer data that is located in the **X** memory. If no memory qualifiers are used, the data is stored into unqualified memory.

For proper integration with the C language, a memory structure is specified, where the unqualified memory encompasses all other memories. All unqualified pointers are pointers into this unqualified memory. The unqualified memory abstraction is needed to keep the compatibility of the **void *** type, the **NULL** pointer, and to avoid duplication of all library code that accesses memory through pointers that are passed as parameters.

NAMED REGISTERS

Embedded C allows direct access to processor registers that are not addressable in any of the machine's address spaces. The processor registers are defined by the compiler-specific, named-register, storage class for each supported processor. The processor registers are declared and used like conventional C variables (in many cases volatile variables). Developers using Embedded C can now develop their applications, including direct access to the condition code register and other processor-specific status flags, in a high-level language, instead of inline assembly code.

Named address spaces and full processor access reduces application dependency on assembly code and shifts the responsibility for computing data types, array and structure offsets, and all those things that C compilers routinely and easily do from developers to compilers.

I/O HARDWARE ADDRESSING

The motivation to include primitives for I/O hardware addressing in Embedded C is to improve the portability of device-driver code. In principle, a hardware device driver should only be concerned with the device itself. The driver operates on the device through device registers, which are device specific. However, the method to access these registers can be very different on different systems, even though it is the same device that is connected. The I/O hardware access primitives aim to create a layer that abstracts the system-specific access method from the device that is accessed. The ultimate goal is to allow source-code portability of device drivers between different systems. In the design of the I/O hardware-addressing interface, three requirements needed to be fulfilled:

1. The device-drive source code must be portable.
2. The interface must not prevent implementations from producing machine code that is as efficient as other methods.
3. The design should permit encapsulation of the system-dependent access method.

The design is based on a small collection of functions that are specified in the <iohw.h> include file. These interfaces are divided into two groups; one group provides access to the device, and the second group maintains the access method abstraction itself.

To access the device, the following functions are defined by Embedded C:

```
unsigned int iord(  ioreg_designator );
void iowr(  ioreg_designator, unsigned int value );
void ioor(  ioreg_designator, unsigned int value );
void ioand( ioreg_designator, unsigned int value );
void ioxor( ioreg_designator, unsigned int value );
```

These interfaces provide read/write access to device registers, as well as typical methods for setting/resetting individual bits. Variants of these functions are defined (with **buf** appended to the names) to access arrays of registers. Variants are also defined (with l appended) to operate with **long** values.

All of these interfaces take an I/O register designator **ioreg_designator** as one of the arguments. These register designators are an abstraction of the real registers provided by the system implementation and hide the access method from the driver source code. Three functions are defined for managing the I/O register designators. Although these are abstract entities for the device driver, the driver does have the obligation to initialize and release the access methods. These functions do not access or initialize the device itself because that is the task of the driver. They allow, for example, the operating system to provide a memory mapping of the device in the user address space.

```
void iogroup_acquire( iogrp_designator );
void iogroup_release( iogrp_designator );
void iogroup_map( iogrp_designator, iogrp_designator );
```

The **iogrp_designator** specifies a logical group of I/O register designators; typically this will be all the registers of one device. Like the I/O register designator, the I/O group designator is an identifier or macro that is provided by the system implementation.The map variant allows cloning of an access method when one device driver is to be used to access multiple identical devices.

EMBEDDED C PORTABILITY

By design, a number of properties in Embedded C are left implementation defined. This implies that the portability of Embedded C programs is not always guaranteed. Embedded C provides access to the performance features of DSPs. As not all processors are equal, not all Embedded C implementations can be equal For example, suppose an application requires 24-bit fixed-point arithmetic and an Embedded C implementation provides only 16 bits because that is the native size of the processor. When the algorithm is expressed in Embedded C, it will not produce outputs of the right precision.

In such a case, there is a mismatch between the requirements of the application and the capabilities of the processor. Under no circumstances, including the use of assembly, will the algorithm run efficiently on such a processor. Embedded C cannot overcome such discrepancies. Yet, Embedded C provides a great improvement in the portability and software engineering of embedded applications. Despite many differences between performance-specific processors, there is a remarkable similarity in the special-purpose features that they provide to speed up applications.

Writing C code with the low-level processor-specific support may at first appear to have many of the portability problems usually associated with assembly code. In the limited experience with porting applications that use Embedded C extensions, an automotive engine controller application (about 8000 lines of source) was ported from the eTPU, a 24-bit special-purpose processor, to a general-purpose 8-bit Freescale 68S08 with about a screen full of definitions put into a single header file. The porting process was much easier than expected. For example, variables that had been implemented on the processor registers were ported to unqualified memory in the general-purpose microprocessor by changing the definitions in the header definition and without any actual code modifications. The exercise was to identify the porting issues and it is clear that the performance of the special-purpose processor is significantly higher than the general-purpose target.

CONCLUSION:

The intelligent traffic light management system proposed in this project offers a practical and effective solution to urban traffic problems. By using IR sensors for traffic density detection and RFID for emergency vehicle prioritization, the system ensures smoother traffic flow and faster emergency response times. It eliminates the drawbacks of traditional fixed-time traffic signals and manual interventions. This project presents a scalable, low-cost, and efficient solution that can play a significant role in the development of smart transportation infrastructure in modern cities.