EMBEDDED DEVICE PROGRAMMING LANGUAGES

- Although there are various programming languages in the embedded programming domain, the vast majority of projects, about 80%, are either implemented in C and its flavors, or in a combination of C and other languages such as C++.
- Some of the striking features of C that aid in embedded development are
- availability of a large number of trained/experienced C programmers,
- support low-level coding to exploit the underlying hardware,
- support in-line assembly code,

EMBEDDED DEVICE PROGRAMMING LANGUAGES

- The ANSI C standard provides customized support for embedded programming.
- support dynamic memory-allocation and recursion,
- provide exclusive access to I/O registers,
- support accessing registers through memory pointers, and
- allow bit-level access.
- compiler support for the vast majority of devices

nesC, Keil C, Dynamic C, and B# are some of the flavors of C used in embedded programming.

nesC

- nesC is a dialect of C that has been used predominantly in sensor-nodes programming.
- It was designed to implement TinyOS an operating system for sensor networks.
- It is also used to develop embedded applications and libraries.
- In nesC, an application is a combination of scheduler and components wired together by specialized mapping constructs.
- nesC extends C through a set of new keywords.
- To improve reliability and optimization, nesC programs are subject to whole program analysis and optimization at compile time.
- nesC prohibits many features that are, function pointers and dynamic memory allocation.
- Since nesC programs will not have indirections, call-graph is known fully at compile time, aiding in optimized code generation.

Keil C

- Keil C is a widely used programming language for embedded devices.
- It has added some key features to ANSI C to make it more suitable for embedded device programming.
- To optimize storage requirements, three types of memory models are available for programmers: small, compact, and large.
- New keywords such as alien, interrupt, bit, data, xdata, reentrant, and so forth, are added to the traditional C keyword set.
- Keil C supports two types of pointers:
- generic pointers: can access any variable regardless of its location
- memory-specific pointers: can access variables stored in data memory

The memory-specific-pointers-based code execute faster than the equivalent code using generic pointers.

This is due to the fact that the compilers can optimize the memory access, since the memory area accessed by pointers is known at compile time

Dynamic C

- Some key features in Dynamic C are
- function chaining and cooperative multitasking.
- Segments of code can be distributed in one or more functions through function chaining.
- Whenever a function chain executes, all the segments belonging to that particular chain execute.
- Function chains can be used to perform data initialization, data recovery, and other kinds of special tasks as desired by the programmer.
- The language provides two directives #makechain, #funcchain and a keyword segchain to manage and define function chains.

Dynamic C

- #makechain chain_name: creates a function chain by the given name.
- #funcchain chain_name func_name[chain_name]: Adds a function or another function chain to a function chain.
- Segchain chain_name {statements}: This is used for function-chain definitions. The program segment enclosed within curly brackets will be attached to the named function chain.
- Dynamic C also has keywords shared and protected, which support data that are shared between different contexts and are stored in battery-backed memory, respectively.

B#

- B# is a multithreaded programming language designed for constrained systems.
- Although C inspires it, its features are derived from a host of languages such as Java, C++, and C#. It supports object oriented programming.
- The idea of boxing/unboxing conversions is from C#.
- For example, a float value can be converted to an object and back to float, as shown in the following code snippet.
- The field property is also similar to C#. B# provides support for multithreading and sync

```
class test{
static void Main(){
        float       i = 123;
        object      obj = i;            // boxing
        float       j = (float)obj;     // Unboxing
        }
}
```

Conclusion of Embedded Programming Languages

- All of the previously described languages have been optimized for resource-constrained devices.
- While designing embedded programs, a measured choice on the flavor of C is quite an important decision from the viewpoint of an IoT programmer.
- An IoT programmer may not restrict himself or herself to a C-flavored language. Many other languages, such as C++, Java, and JavaScript have been stripped down to run on embedded devices.

MESSAGE PASSING IN DEVICES

- some of the communication paradigms and technologies are used in Message Passing. They are RPC, REST, and CoAP .
- They can be used in resource-constrained environments.

RPC: is an abstraction for procedural calls across languages, platforms, and protection mechanisms.

- For IoT, RPC can support communication between devices as it implements the request/response communication pattern.
- When RPC messages are transported over the network, all the parameters are serialized into a sequence of bytes.
- Since serialization of primitive data types is a simple concatenation of individual bytes, the serialization of complex data structures and objects is often tightly coupled to platforms and programming languages. This strongly hinders the applicability of RPCs in IoT due to interoperability concerns.

MESSAGE PASSING IN DEVICES
Lightweight Remote Procedure Call (LRPC)

- was designed for optimized communication between protection domains in the same machine, but not across machines.
- Embedded RPC (ERPC) uses a fat client such as a PC and thin servers such as nodes architecture.
- This allows resource-rich clients to directly call functions on applications in embedded devices.
- S-RPC is another lightweight remote procedure-call for heterogeneous WSN networks. S-RPC tries to minimize the resource requirements for encoding/decoding and data buffering.
- A trade-off is achieved based on the data types supported and their resource consumption. Also, a new data representation scheme is defined which minimizes the overhead on packets. A lightweight RPC has been incorporated into the TinyOS, nesC environment. This approach promises ease of use, lightweight implementation, local call-semantics, and adaptability.