Congr. Int. en Ing. Electrónica. Mem. ELECTRO, Vol. 43 pp. 50 - 55, Oct 2021, Chihuahua, Chih. México
http:// electro.itchihuahua.edu.mx/revista
ISSN 1405-2172

USO DE PATRONES DE DISEÑO Y METAPROGRAMACIÓN PARA CONSTRUIR APIS DE IOT USANDO C++

Alberto Pacheco, Jesús Escobar, Edgar Trujillo

Instituto Tecnológico de Chihuahua
División de Estudios de Posgrado e Investigación
Av. Tecnológico 2909. Col. 10 de Mayo, Chihuahua, Chih. CP 31310
{alberto.pg, m20061556, edgar.tp}@chihuahua.tecnm.mx

RESUMEN

Ante el explosivo crecimiento del internet de las cosas, resulta imperativo desarrollar software embebido para ecosistemas cada vez más complejos y heterogéneos. La metaprogramación y diseño de patrones permiten reusar componentes para diferentes plataformas, sin embargo, la eficiencia requerida para un sistema IoT puede limitar su uso. El presente trabajo demuestra que es posible diseñar APIs para IoT suficientemente genéricas, robustas, portables y eficientes aplicando la metaprogramación con *templates* de C++. Para su desarrollo se aplicó un proceso iterativo (tres ciclos) de investigación-acción validado bajo diferentes plataformas, dispositivos, sensores y actuadores logrando un código más compacto y un rendimiento equiparable a la versión implementada en lenguaje C.

Palabras Clave: Internet de las Cosas; Metaprogramación; Diseño de APIs; Patrones de diseño; Software embebido; Investigación-acción.

ABSTRACT

The explosive growth of the internet of things urges the need of embedded software best adapted to increasingly complex and heterogeneous ecosystems. Metaprogramming and design patterns allow the reuse of components for different platforms, however the efficiency required for an IoT system can limit its possible practical use. This work shows a generic, robust, portable and efficient API for IoT that applied C++ template metaprogramming, developed through an iterative action-research process validated in different platforms, devices, sensors and actuators, performing as well as its C version.

Keywords: IoT; Metaprogramming; API design; Design patterns; Embedded software; Action-research.

1. INTRODUCCIÓN

Una interfaz de programación de aplicaciones (API) define cada componente como un conjunto de funcionalidades independientes de sus detalles de implementación [1-2], y se caracteriza por: a) abstraer entidades y comportamientos; b) diseñar y documentar teniendo en mente al usuario; c) ser modular, consistente, estable, confiable, flexible, reusable y fácil de adoptar. El objetivo del presente trabajo es explorar la incorporación de patrones de diseño y metaprogramación en el desarrollo de un API para la capa de percepción de un sistema de telemetría IoT compuesto de nodos heterogéneos en red para la adquisición, fusión y transmisión de lecturas de sensores, control de actuadores y despliegue de datos en dispositivos móviles (Fig. 1).

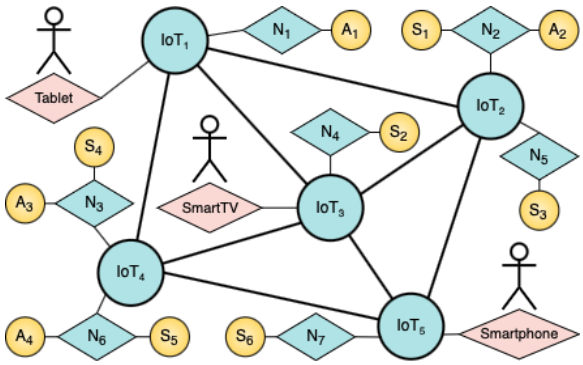

Fig. 1. *Ecosistema IoT con nodos y sensores heterogéneos.*

Diseño de APIs para IoT. El desarrollo de software embebido antepone la eficiencia de máquina, manejando APIs muy específicas, eficientes y de bajo nivel escritas por lo general en lenguaje C y ensamblador (arco ⓐ en Fig. 2). Con los avances en la tecnología y el crecimiento en la complejidad de los sistemas heterogéneos IoT, ha surgido un interés por el diseño de APIs cada vez más genéricas, robustas y adaptables. Aunque la programación orientada a objetos (OOP) ayuda a modelar sistemas complejos (arco ⓑ Fig. 2), es impopular en sistemas embebidos de limitada capacidad donde, en ciertos casos, puede consumir más memoria y reducir el desempeño [3].

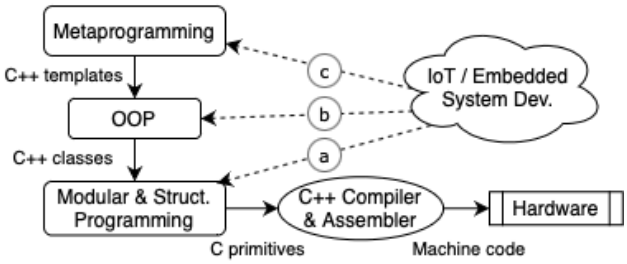

Fig. 2. *Paradigmas de desarrollo de software embebido.*

Metaprogramación en IoT. Mediante plantillas o clases parametrizadas de C++ (arco ⓒ Fig. 2), es posible aplicar la metaprogramación (código que genera código) y la

programación genérica [4] sin penalizar el desempeño de un programa (*zero-cost abstraction*) [5]. Un *template* de C++ permite que una misma función o clase trabaje con distintos tipos de datos, indicándolos como parámetros (*template parameters*) para instanciar el código específico para dicha clase o función (*template specialization*) [6], con la ventaja de que el programador puede definir reglas para controlar dicho mecanismo de especialización (*Turing-complete system*) [4].

2. TRABAJOS RELACIONADOS

Si bien se localizaron pocos trabajos sobre la aplicación de la metaprogramación en IoT [5-8], en [7] se presentan diversas técnicas de metaprogramación, UML, patrones de diseño y metamodelos en sistemas embebidos, destacando el diseño de alto nivel en hardware y software enfocado en el reuso (d*esign-for-reuse*), la creciente complejidad en sistemas en un chip (*SoC*), la incipiente adopción de UML y C++ en sistemas embebidos. Este tema se aborda en P0568 por el comité SG14 para el estándar de C++, así como en diversos cursos [9] y congresos: emBO++, PLoP, Cppcon y *Embedded online conference*. Por su parte, en [10] se presentan avances y retos sobre el paradigma emergente de comunicación entre dispositivos (D2D) con redes 5G abarcando áreas de aplicación como IoT, IoVT (*vehicular IoT*), IoMT (*medical IoT*).

Arquitectura de IoT. Un sistema de telemetría consiste en el monitoreo en tiempo real de magnitudes físicas y la transmisión de dichas variables, con el fin de controlar un sistema remoto. En la arquitectura IoT de tres capas [11] (Fig. 3): 1) la capa de percepción es un conjunto de nodos IoT con actuadores y sensores que interactúan con el entorno físico; 2) la capa de red abarca nodos capaces de fusionar y transmitir datos mediante enlaces de red; 3) la capa de aplicación interactúa con los usuarios y depende del dominio y problema a resolver. El presente trabajo aborda exclusivamente la capa de percepción, el desarrollo del resto de las capas se aborda en [12].

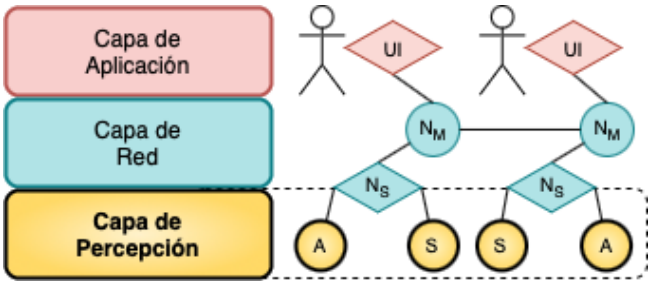

Fig. 3. *Arquitectura IoT de tres capas [11].*

3. OBJETIVO Y METODOLOGÍA

Objetivo de diseño del API para IoT. El objetivo de diseño del API para la capa IoT de percepción (Fig. 3) es facilitar el desarrollo de aplicaciones IoT de telemetría usando nodos y dispositivos heterogéneos, siendo capaz de abstraer la capa física mediante un API genérica, robusta, eficiente y amigable.

Desarrollando en C++ para IoT. Para el desarrollo del API se utilizó el entorno Arduino (IDE), una plataforma de hardware abierto popular entre aficionados y estudiantes, que soporta modelos básicos y accesibles (Uno, Mega, Zero), desde un ATmega328 (8 bits, 20 MHz, 20 MIPS) hasta modelos recientes como Arduino Nano 33 (Tabla 1). Para este trabajo fue elegida la tarjeta Espressif ESP32, que alcanza los 600 MIPS a 240 MHz (Tabla 1). El SoC ESP32 que tiene un costo inferior al Arduino Uno, cuenta con WiFi y Bluetooth, siendo capaz de correr modelos ML, Rust, FreeRTOS o funciones tipo Alexa [13]. Además, soporta lenguaje C++11, ofreciendo una alternativa interesante para implementar el API propuesta.

Tabla 1. Tarjetas Arduino Nano 33 y ESP32.

Company Model	MCU bits/cores	Clock speed RAM/Flash	C++ Compiler	GPIO-SPI-I2C-UART-ADC (RF)	Cost US$
Arduino Nano 33 BLE Sense	ARM Cortex-M4F Nordic nRF52840 32-bit 1-core	64 MHz 256K / 1M	Arduino avr-g++ 7.x	14-1-1-1-8-12b (BLE 5)	$20-$35
Espressif ESP32	Xtensa LX6 32-bit 2-cores	240 MHz 320K+ / 4M	Xtensa GNU C++11	34-2-2-2-12b (WiFi / BLE)	$15-$30

Caso de uso. En particular, para la capa IoT de percepción, existen un conjunto de nodos-esclavo con diversos sensores y actuadores (nodos Ns en Fig. 3). Por otro lado, los nodos-maestro (nodos Nm en Fig. 3) fusionan los datos provenientes de diversos nodos, mismos que pueden replicar a otros nodos y dispositivos móviles de usuarios (nodos UI en Fig. 3) [12]. A continuación, se describen los nodos utilizados.

Nodo IoT esclavo. La tarjeta de bajo costo Espressif ESP32 permite enlazar vía Bluetooth con nodo maestro, adquirir datos de un sensor de temperatura y manejar relevadores (Fig. 4).

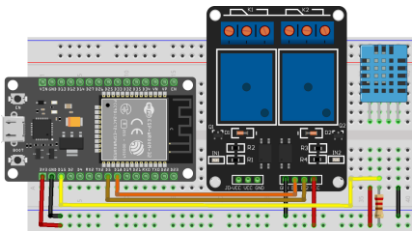

Fig. 4. *Nodo-esclavo con sensores y actuadores.*

Nodo IoT maestro. Se uso la tarjeta Heltec ESP32 520K RAM, 8M Flash, pantalla OLED 0.96” 128x64 (Fig. 5), con un reloj de tiempo real RTC DS1302 (vía 3 puertos GPIOs) y teclado matricial (4 GPIOs). Este nodo se enlaza a nodos-esclavo vía Bluetooth y a nodos-maestro vía LoRa. Además, interactúa con el usuario vía pantalla y teclado.

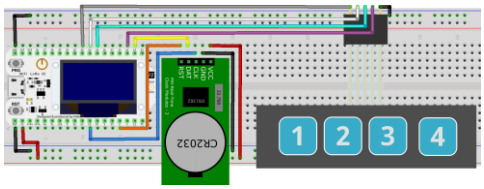

Fig. 5. *Nodo-maestro con reloj, pantalla y teclado.*

Un caso de uso. Para ilustrar las metas de diseño del API, considerando los nodos IoT mencionados, se describe enseguida un caso de uso basado en un sensor de temperatura (líneas 1, 3) y tres relevadores (línea 2). En el ciclo principal, cuando se rebasa cierto nivel de temperatura (línea 5), los actuadores y el indicador LED se activan usando una simple asignación (línea 6). Además, se genera y envía un paquete de datos al nodo-maestro (línea 7). El presente artículo se centra en describir el desarrollo iterativo del API aplicando la metodología de investigación-acción, patrones de diseño y meta-programación, presentando las diversas versiones y pruebas realizadas para la capa de percepción (Fig. 3).

```
1: TempSenT  heat;
2: RelayN relays(PINS(16,17,18));
  …
3: heat.begin(PINS(14,15));
  …
4: void loop() {
5:  if (heat.sense() > 33.0) {
6:    BuiltinLED = relays = ON;
7:    BLE_Node.send(heat.packet()); …
```

Metodología Investigación-Acción (AR). Se aplicó diseño de patrones con diagramas UML [14] y la metodología AR, misma que se enfoca a buscar una solución comprendiendo primero el problema de forma iterativa, pragmática y participativa [15-16]. En cada ciclo AR, los investigadores de forma activa y participativa establecen un plan de acción, se ejecuta (acción), evalúa y reflexiona sobre lo aprendido para incluir mejoras en el siguiente ciclo [17]. Se efectuaron tres ciclos-AR para desarrollar y evaluar de forma participativa un API para la capa de percepción que fuera a la vez eficiente y genérica.

4. DISEÑO Y DESARROLLO DEL API

Ciclo AR1: Primer prototipo. En el primer ciclo AR, se identificaron las entidades y operaciones básicas y se definió un plan para implementar las estructuras y funciones aplicando manejadores de dispositivos como patrón de diseño [18]. Se implementó en C, usando estructuras (líneas 10-13), apuntadores (*handlers*) y constantes para tipo de sensor/actuador (líneas 14-15):

```
10: ActionG  r1, r2, r3;
11: ControlA rel1, rel2, rel3;
12: SensorG  t1;
13: ControlS heat;
    …
14: sense_begin(&heat, &t1, TEMP, Pin(15));
15: act_begin(&rel1, &r1, RELAY, Pin(16));
    …
16: void loop() {…
17:   if (sense(&heat) > 33.0) {
18:    control_cmd(&rel1, ON);
19:    control_cmd(&rel2, ON); …
```

Ciclo AR2: Polimorfismo. Reflexionando sobre las limitantes y lecciones aprendidas en el ciclo anterior, al descubrir que el compilador soportaba C++11, se decidió aplicar la programación orientada a objetos usando clases y funciones virtuales. Aplicando los patrones de diseño *Composite* (`ControlA` *has* `ActionG`) y *Adapter* (`ActionRelay`, `ActionLED`) [14], se estableció en la clase `ControlA` una interfaz genérica para un actuador (Fig. 6). Como se muestra en la clase del actuador genérico `ActionG`, es posible agregar nuevos tipos de actuadores derivado dicha clase respetando una misma interfaz para inicializar y operar un actuador (métodos `begin` y `cmd` en Fig. 6). Estos mismos patrones de diseño fueron aplicados para implementar los sensores y sus controladores.

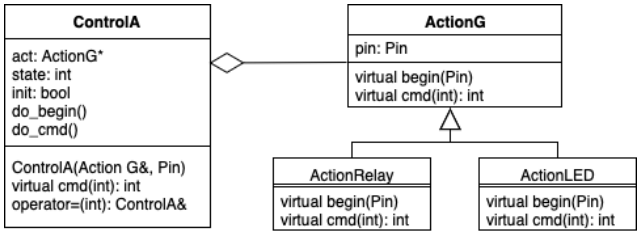

Fig. 6. *Controlador y actuador polimórfico.*

El polimorfismo en C++ se logra con un apuntador a una clase-base (línea 25) para luego invocar una misma acción en cualquier actuador (línea 27). Esta versión satisface las metas de diseño propuestas (sección 3). Además, gracias a la sobrecarga de operadores en C++ (línea 28), es posible operar actuadores asignando valores (línea 6). Sin embargo, este API sólo soporta operaciones con un sólo tipo de valor para sensor (`float`), actuador (`int`) y conexiones con un sólo puerto (`Pin`).

```
20: class ActionG {
21:   virtual int cmd(int) = 0; …
22: class ActionRelay : public ActionG {
23:   virtual int cmd(int c) { … } …
24: class ControlA {
25:   ActionG* act; …
26:   ControlA(ActionG &r, Pin p) { … }
27:   int cmd(int c) {… act->cmd(c); }
28:   int operator= (int c) {… cmd(c); } …
```

Ciclo AR3: Metaprogramación. Generando clases a partir de *templates* (metaprogramación), es posible definir un *template* `SensorG<T,N>` (Fig. 7) con parámetros para designar el tipo de

valor para un sensor específico (tipo `T`) con la cantidad precisa de puertos requeridos `Pins<N>` (*template specialization*). Dentro del *template* `SensorG`, el parámetro `T` define el tipo del método `sense` (línea 33) y el parámetro `N` el tipo del atributo `pins` (línea 31) y el tipo de argumento de `begin` (línea 32). Para derivar un nuevo tipo de sensor con `T1=char` y `N=4` puertos de conexión de un teclado matricial, usamos `SensorK4<char,4>` (Fig. 7). Este mismo modelo se aplicó para los actuadores.

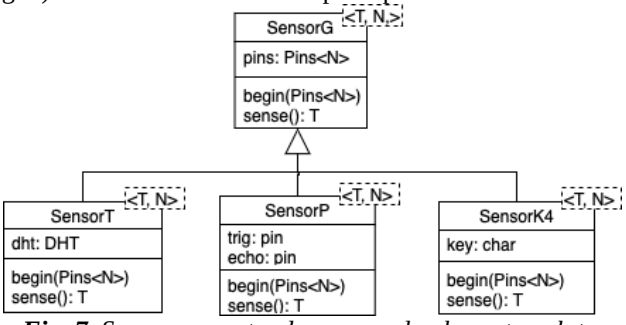

Fig. 7. *Sensores y actuadores usando clases-template.*

```
29: template<class T, int N>
30: struct SensorG {
31:    private: Pins<N> pins;
32:    void begin(Pins<N> p) { pins=p; }
33:    T sense() { … }
```

Y para `Pins` usando *template* de sinónimo de tipo tenemos:

```
34: template<int N>
35:   using Pins = std::array<byte, N>;
```

Lo que permitió cambiar la implementación de `Pins` sin alterar el resto del API y probar distintas estructuras (`tuple, vector, array`). Resultando `array` con tamaño fijo, un código más eficiente y familiar. Al generar los puertos de un sensor (línea 3), basta usar `Pins<2> {14,15}`, para generar un `array<byte,2>` al llamar al método `begin` del template `SensorG` (línea 32), mismo que asigna al atributo `pins` de tipo `Pins<N>` (línea 31).

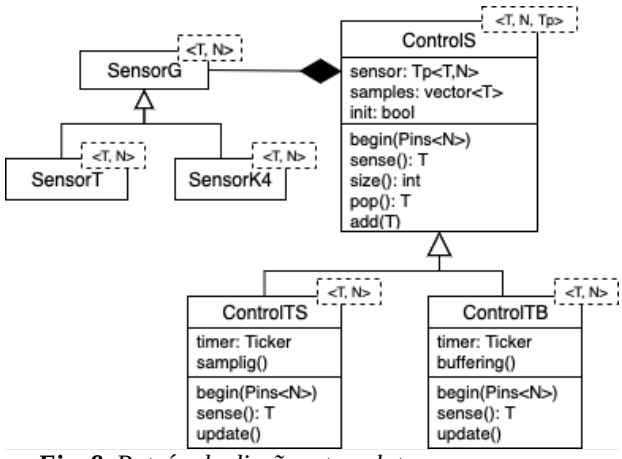

Fig. 8. *Patrón de diseño y templates para sensores.*

Controladores de sensores y actuadores. Para lograr un diseño más genérico para los actuadores y sensores, se aplicó el patrón de diseño de composición (*Composite)* [14], donde un controlador maneja actuadores o sensores (Fig. 8). Esto permite reusar código, por ejemplo, para inicializar, validar, temporizar lecturas, manejar y transmitir datos, y el manejo de errores.

Para los sensores, el *template* genérico `ControlS` soporta sensores básicos, y puede especializarse para muestrear lecturas (`ControlTS`) o para almacenar lecturas en un *buffer* (`ControlTB`). Sus parámetros `T`, `N` y `Tp` definen el tipo de sensor a manejar (línea 27) que será inicializado con `begin` (línea 30); tanto la colección de muestras (línea 28), como su lectura (línea 31) se especializan para valores tipo `T`. A su vez, el controlador ofrece funciones para administrar colecciones de muestras (`size`, `empty`, `remove`) (Fig. 8 y línea 32).

```
25: template<class T, int N, template… Tp>
26: class ControlS {
27:    Tp<T,N> sensor;
28:    std::vector<T> samples;
29:  public:
30:    void begin(Pins<N> p) {… sensor.begin(p); …}
31:    T sense() { … }
32:    int size()  { … } …
```

Por otro lado, `ControlTS` hereda toda la funcionalidad de la metaclase `ControlS` (líneas 33-34). Este nuevo tipo de controlador para sensores adiciona un temporizador `Ticker` para tomar muestras de manera asíncrona (líneas 35 y 38), de tal forma que al solicitar el valor actual del sensor, se calcula el valor promedio de las muestras (línea 39).

```
33: template<class T, int N, template… Tp>
34: class ControlTS : public ControlS<T,N,Tp> {
35:    Ticker timer;
36:  public:
37:    void begin(Pins<N> p) { … }
38:    void update() { … }
39:    T sense() { return accumulate(…)/size(); }
```

Múltiples actuadores. Para implementar los actuadores, se aplicaron los mismos patrones de diseño y técnicas de metaprogramación usados para los sensores. Sin embargo, destaca la activación de múltiples actuadores para el caso de uso (línea 6), que si bien puede aplicarse un ciclo para operar una colección de actuadores de la siguiente forma:

```
40: BuiltinLED.cmd(OFF);
41: for (int i=0; i<relays.size(); i++)
42:   relays.cmd(OFF, i);
```

Para el template para múltiples actuadores `ControlAN` se aplicó la sobrecarga del operador de asignación `operator=`

(línea 46) que invoca la operación de múltiples actuadores (línea 45), logrando así implementar el caso de uso como aparece en la línea 7.

```
43: template <class T, int N, template… Tp>
44: class ControlAN: public ControlA<T,N,Tp> {
45:  inline T cmd_all(T c) { … }
46:  inline T operator=(T c) {return cmd_all(c);}
```

Sinónimos de tipo para sensores y actuadores. Aunque es posible definir nuevos sensores y actuadores, para facilitar la programación se definieron sinónimos de tipos para los casos más comunes. Por ejemplo, el sensor básico de temperatura `TemSense` usando un puerto, `ControlS` y `SensorT` (línea 47). `TempSenT` muestrea de manera asíncrona usando un *timer* (línea 48). `Key4Sense` proporciona un teclado matricial asíncrono temporizado con *buffer* conectado vía 4 puertos (línea 49). Por último, es posible tener un controlador `RelayN` que administre `N` relevadores/puertos (líneas 50-51), e.g. para manejar dos relevadores en los puertos 4 y 5, se declara como `RelayN<2> relays(PINS(4,5));`

```
47: using TempSense=ControlS<float,1,SensorT>;
48: using TempSenT=ControlTS<float,1,SensorT>;
49: using Key4Sense=ControlTB<char,4,SensorK4>;
50: template<int N>
51:  using RelayN=ControlAN<bool,N,ActionRelay>;
```

Extendiendo y refinando el API. Finalmente, para ilustrar como incorporar nuevos tipos de actuadores o sensores, se presentan como ejemplo, el teclado hexadecimal `HexPad` basado en puerto I2C y teclado `Key16` (línea 52) y un teclado matricial `EmoPad` con cuatro caracteres `Emoji` reusando `SensorK4` (línea 53). Ambos casos reusan el controlador de buffer genérico `ControlTB`. Lo mismo aplica para el caso en que sea necesario definir nuevos sensores.

```
52: using HexPad = ControlTB<char, I2C, Key16>;
53: using EmoPad = ControlTB<Emoji, 4, SensorK4>;
```

5. PRUEBAS Y VALIDACIONES

Metaclases con restricciones. Para limitar a un rango específico de puertos en el *template* `SensorG`, tenemos:

```
54: template<typename T1, int N>
55: class SensorG {
56:   static_assert(N>0 && N<=8,"Error …"); …
```

Mediante `static_assert()` se verifica en tiempo de compilación si el parámetro `N` está en rango (línea 56). Por ejemplo, `SensorG<float,-1> heat;` generará el siguiente error:

```
TestSensor.ino … instantiation SensorG<float, -1>
iot_sensor.h … assertion failed: Error N is out of range
    static_assert(N>0 && N<=8, …
```

Además, se verifica que el parámetro `T` sea tipo `char` con `is_same` en el *template* `SensorK4` (línea 59). Si por ejemplo se tiene `SensorK4<float,4>`, marcará error (`T=float`).

```
57: template<typename T, int N>
58: class SensorK4 : public SensorG<T1,N> {
59:   static_assert(is_same<T,char>,"Error…");…
```

Técnicas avanzadas de Metaprogramación. Para establecer múltiples puertos (*pins*), se aplicó el patrón de diseño *Factory Method* [14], definiendo una función-*template* con un número variable de parámetros (*variadic template*) indicado con puntos suspensivos `...` (*parameter pack*) [19]:

```
60: template <class... T>
61: auto PINS(T... args) {
62:  static_assert(
63:   conjunction_v<is_integral<T>...>,"Error…");
64:  return Pins<sizeof...(T)> {
65:   static_cast<byte>(args)...};
66: }
```

A diferencia de una macro de lenguaje C (ciclo AR1), la función-*template* `PINS` valida que cada parámetro sea entero mediante `conjunction` / `is_integral` (línea 63); en caso de fallar, genera error (líneas 62-63). Si es correcto, genera una instancia del *template* `Pins` obteniendo el número de parámetros recibidos con `sizeof...` (línea 64) inicializada con argumentos convertidos a bytes usando `static_cast` (línea 65). Además, el compilador infiere con `auto` (línea 61) el tipo de la función-template `PINS`. De esta forma, `PINS(12,13,14)` genera una instancia con tres puertos y para `PINS(true,1.7)` mostrará un error.

Validación en tiempo de ejecución. Para implementar un API robusta es necesario también atender fallas por condiciones que lleguen a presentarse cuando se ejecuta el programa. Ejemplo:

```
67: TempSenT heat;
68: auto temp = heat.sense();
```

Donde se usa un sensor no inicializado (línea 68). Como se observa abajo, el `ASSERT` detecta y despliega un mensaje de error (línea 71) cuando no se inicializa con `begin` (línea 72). Para acelerar dicha verificación (*RTTI-overhead*), se declararon `inline` los métodos de la clase-*template* (líneas 71-73).

```
69: template<typename T1, …>
70: class ControlS {
71:   inline void do_read() {ASSERT(init,"Err…");}
72:   inline void begin(…) { init=true; … }
73:   inline T1   sense() { … } …
```

6. ANÁLISIS DE RESULTADOS OBTENIDOS

Se analizó el código generado en las tres versiones (ciclos AR) usando tres compiladores distintos (Tabla 2): Arduino GNU C++11 (ARM 32 bits), Xcode 12.5 con GNU++ (iOS 64 bits) y Compiler Explorer GCC (ARM 32 bits) [21]. Analizando y comparando las tres versiones (ciclos AR), la misma sentencia de activación de actuadores del caso de uso (línea 6). En la versión de lenguaje C (ciclo AR1), hay cinco niveles de invocación de funciones, de los cuales dos son macros. Como se aprecia en su implementación (líneas 10-19), el API resulta compleja por el manejo de apuntadores, las funciones de inicialización tienen muchos parámetros, los apuntadores pueden inducir a errores de codificación, además del uso de constantes para identificar el tipo de sensor/actuador obliga a manejar sentencias `switch`, reduciendo el desempeño y costo de mantenimiento, sobre todo cuando se agregue un nuevo sensor/actuador.

En la versión OOP de C++98 (ciclo AR2), la sentencia de activación (líneas 6 y 80) generó cinco niveles de llamada: el operador de asignación, dos funciones virtuales `cmd`, función `out` (relay) y función `digitalWrite` (puerto). Esta versión ofrece un API simple, consistente, pero limitado, debido al perfil fijo de una función virtual (polimorfismo estático), sólo soporta un tipo de comando (línea 80), lecturas tipo `float` en sensores (línea 79), un actuador por controlador (línea 76) y no soporta funciones `inline`. Su caso de uso quedó como sigue:

```
74: TempSenT heat;
75: ActionRelay r, s, t;
76: Relay a(r,pin(16)),b(s,pin(17)),c(t,pin(18));
77: heat.begin(pin(14));
   …
78: void loop() {
79:  if (heat.sense() > 33.0) {
80:    BuiltinLED = a = b = c = ON; …
```

La versión final implementada en C++17 (ciclo AR3), coincide con el caso de uso propuesto (líneas 1-7). Para la línea 6 del caso de uso, hay seis niveles de invocación: 1) operador de asignación; 2) función `cmd_all` del controlador; 3) función `cmd_all` del actuador; 4) ciclo del `array`; 5) función `out`; 6) función `digitalWrite`. Gracias a las funciones `inline`, excepto por las llamadas para librerías externas (`array` y Arduino `digitalWrite`), se eliminó el costo de llamada de 5 funciones. El código `inline` generado fue:

```
81: auto c=ON, relay_pins=array<byte,3> {16,17,18};
82: for (auto& p : relay_pins) // cmd_all(c)
83:   digitalWrite(p, c? HIGH : LOW); // out(p,c)
```

La Tabla 2 muestra los tamaños de compilación para las líneas 2 y 6 del caso de uso en cada ciclo AR, tanto sin optimizar (`-O0`) como optimizada (`-O2`). La versión con *templates* fue la de menor tamaño en todas las opciones y la versión OO con funciones virtuales sin optimizar fue la de mayor tamaño.

Tabla 2. Resultados caso de uso por ciclo AR usando [21]

Ciclo	Compilador	Técnica(s)	Tamaño (-O0) sin/con inline	Tamaño (-O2) sin/con inline
AR1	C99, gcc 4.5	Macros & funciones	1,616 / 1,576	1,040 / 776
AR2	C++98, gcc 4.5	Herencia & f. virtual	1,804 / ---	892 / ---
AR3	C++17, gcc 11	Metaprog. template	1,722 / 1,384	530 / 530

7. CONCLUSIONES

Aplicando patrones de diseño, metaprogramación con *templates* de C++ se logró una implementación robusta, eficiente, genérica y portable (ARM 32/64 bits) de un API para la capa de percepción de un sistema IoT. Los *templates* se especializaron en tiempo de compilación de acuerdo a las entidades requeridas en la aplicación (caso de uso), sin penalizar el rendimiento del sistema. A futuro, se espera aplicar las lecciones aprendidas en el desarrollo de la capa de comunicación en red, así como la revisión de técnicas más avanzadas de metaprogramación (*type erasure, concepts, lambdas*). Por último, se pretende seguir explorando técnicas de metaprogramación en el desarrollo de sistemas IoT complejos y en la impartición de cursos de programación C++ para IoT incluyendo otros entornos.

Agradecimientos: Este trabajo forma parte del proyecto "*Propuesta de red LoRA tipo malla de bajo costo aplicable a la telemetría del sector Industria 4.0"* auspiciado por el Tecnológico Nacional de México.

Referencias

[1] M. Reddy, *API Design for C++*, Morgan Kaufmann, 2011.
[2] M. Biehl, *API Architecture: The Big Picture for Building APIs*, CreateSpace, 2015. ISBN: 9781508676645
[3] A. Chatzigeorgiou & G. Stephanides, Evaluating performance and power of OOP vs. procedural prog. in embedded processors, en *Reliable Soft Tech*, 2361(1):65–75, Springer, 2002. doi: 10.1007/3-540-48046-3_5
[4] E. Alligand & J Falcou, *Practical C++ Metaprogramming*. O'Reilly, 2016. ISBN: 9781491955031
[5] B. Andrist & V. Sehr, *C++ High Performance: Boost and optimize the performance of your C++17 code*, Packt, 2017.
[6] D. Vandevoorde & N. Josuttis, *C++ Templates: The Complete Guide*, Pearson, 2002. ISBN: 9780672334054
[7] R. Damasevicius & V. Stuikys, High-Level Models for Transformation-Oriented Design of Hardware and Embedded Systems, en *Adv Electrical and Computer Eng*, 8(2):86-94, 2008, doi: 10.4316/AECE.2008.02016
[8] W. Raschke, et al., Patterns for hardware-independent development for embedded systems, en *Proc. Euro. Conf. on Pattern Lang. Programs*, 1–14, 2014. doi: 10.1145/2721956.2721959
[9] S. Meyers, *Effective C++ in an Embedded Environment*, Artima, 2015.
[10] C. Chakraborty et al., A comprehensive review on device-to-device com. paradigm: trends, challenges and apps, en *Wireless Pers Commun*, 114:185–207, 2020. doi: 10.1007/s11277-020-07358-3
[11] P. Sethi, & S. Sarangi, Internet of Things: Architectures, Protocols, and Applications, en *J. Electrical and Computer Eng.*, 1-25, 2017
[12] J. Escobar, *Red Tipo Malla Propagable para Telemetría en IoT* (tesis en desarrollo), TecNM IT Chihuahua, 2021.
[13] TensorFlow, TensorFlow Lite para microcontroladores, 2021. doi: 10.5281/zenodo.4724125
[14] E. Gamma, et al., *Patrones de Diseño.* Addison Wesley, 1995.
[15] K. Petersen, et al., Action research as a model for industry-academia collaboration in the software engineering context, en *ACM Proc. Int. Works. Ind. Collab. on SE*, 2014, 55–62, doi: 10.1145/2647648.2647656

[16] M. Staron, Action Research as Research Methodology in Software Engineering, en Action Research in Software Engineering, Springer, 2020
[17] N. Surendra & S Nazir, Creating info systems using Agile dev practices: an action research study, en *Euro. J. Info Sys*, 28(5):549-565, 2019
[18] C. Preschern, API patterns in C, en *ACM Proc. Euro. Conf. on Pattern Lang. Programs*, pp. 1–11, 2016. doi: 10.1145/3011784.3011791
[19] D. Nesteruk, *Design patterns in modern C++*. Apress, 2018.
[20] N. Josuttis, *C++17 - La guía completa* (trad. J. Estrada). LeanPub, 2020.
[21] Compiler Explorer on-line C++ compiler, 2021. https://godbolt.org