Serializar y deserializar instrucciones

¡Excelente! Has subido el manual de referencia de RAPID, el lenguaje de programación de los robots ABB. Este es un caso de estudio perfecto para un serializador/deserializador.

Analizando el documento, estamos ante un lenguaje industrial robusto con:

· Tipos de datos complejos: robtarget (posición + orientación + configuración + ejes externos), jointtarget, speeddata, zonedata, etc.

· Instrucciones con sintaxis variada: Desde simples (Set do1;) hasta complejas con múltiples argumentos opcionales y mutuamente excluyentes (MoveL [\Conc] ToPoint Speed [\V] | [\T] Zone [\Z] Tool [\WObj]).

· Estructura modular: Módulos, rutinas, datos.

Tu idea es correcta: generar el deserializador automáticamente a partir de la especificación de las instrucciones es el enfoque más mantenible y escalable. Te propongo una arquitectura para abordar este problema.

Arquitectura Propuesta: Enfoque de Metaprogramación

Dividiremos el problema en tres componentes principales:

1. Un Especificador de Instrucciones (Formato Intermedio): Definir un DSL (Domain Specific Language) en JSON/YAML que capture la estructura de cada instrucción de RAPID.

2. Un Generador de Código: Un script (en Python, por ejemplo) que lea ese DSL y genere el código del serializador/deserializador en tu lenguaje objetivo (C, C++, Rust, Python, etc.).

3. La Librería de Serialización/Deserialización: El código generado que hace el trabajo pesado, apoyándose en una capa base para leer/escribir bytes y manejar los tipos de datos.

Paso 1: Definir el Formato Intermedio (DSL)

Necesitamos una forma de describir cada instrucción. Usaré JSON como ejemplo por su claridad.

```json
{
  "instructions": [
    {
      "name": "MoveL",
      "description": "Moves the robot linearly",
      "syntax": "MoveL [\\Conc] ToPoint Speed [ \\V ] | [ \\T ] Zone [ \\Z ] Tool [ \\WObj ]",
      "opcode": 0x01,
      "arguments": [
        { "name": "Conc", "type": "switch", "optional": true, "tag": 0x01 },
        { "name": "ToPoint", "type": "robtarget", "required": true },
        { "name": "Speed", "type": "speeddata", "required": true },
        { "name": "V", "type": "num", "optional": true, "group": "speed_mode", "tag": 0x02 },
        { "name": "T", "type": "num", "optional": true, "group": "speed_mode", "tag": 0x03 },
        { "name": "Zone", "type": "zonedata", "required": true },
```

```
      { "name": "Z", "type": "num", "optional": true, "tag": 0x04 },
      { "name": "Tool", "type": "tooldata", "required": true },
      { "name": "WObj", "type": "wobjdata", "optional": true, "tag": 0x05 }
    ]
  },
  {
    "name": "Set",
    "opcode": 0x02,
    "arguments": [
      { "name": "Signal", "type": "signaldo", "required": true },
      { "name": "Value", "type": "dionum", "required": true }
    ]
  }
  // ... instrucciones para IF, FOR, WHILE, etc.
 ]
}
```

Puntos clave de este DSL:

· opcode: Un byte único que identifica la instrucción.

· group: Maneja argumentos mutuamente excluyentes (ej. \V o \T). En la serialización, primero se escribe un byte que indica qué argumento del grupo está presente.

· tag: Un identificador (ej. un byte) para argumentos opcionales, permitiendo un formato tipo TLV (Type-Length-Value).

· type: Puede ser un tipo primitivo (num, bool, string) o un tipo compuesto (robtarget). Estos tipos compuestos también necesitarán su propia definición en el DSL.

Paso 2: Definir los Tipos de Datos Compuestos

Los tipos como robtarget, speeddata, etc., tienen una estructura fija. Debemos definirlos también.

```json
{
  "data_types": [
    {
      "name": "robtarget",
      "fields": [
        { "name": "trans", "type": "pos" },
        { "name": "rot", "type": "orient" },
        { "name": "robconf", "type": "confdata" },
        { "name": "extax", "type": "extjoint" }
      ]
    },
    {
      "name": "speeddata",
      "fields": [
        { "name": "v_tcp", "type": "num" },
        { "name": "v_ori", "type": "num" },
        { "name": "v_leax", "type": "num" },
```

```
      { "name": "v_reax", "type": "num" }
    ]
  }
  // ... definiciones para pos, orient, confdata, extjoint, etc.
 ]
}
```

Paso 3: El Generador de Código

El generador leerá estos JSONs y producirá código en tu lenguaje de destino. A grandes rasgos, hará:

1. Para cada tipo de dato (robtarget, speeddata, etc.):
   · Generar una función serialize_Tipo() que escribe cada campo en orden en un buffer.
   · Generar una función deserialize_Tipo() que lee el buffer y reconstruye la estructura.
2. Para cada instrucción:
   · Generar una función serialize_Instruccion() que:
     1. Escribe el opcode.
     2. Escribe los argumentos requeridos en orden.
     3. Para argumentos opcionales, escribe un byte de máscara de bits indicando cuáles están presentes, seguido de los datos de cada uno.
     4. Para grupos mutuamente excluyentes, escribe un byte indicando la opción seleccionada y luego sus datos.
   · Generar una función deserialize_Instruccion() que haga lo inverso.

Paso 4: Formato de Serialización Binaria Propuesto

Dado que es para una máquina virtual, un formato binario es ideal.

```
+------------------+------------------+------------------+------------------+
| Opcode (1 byte) | Flags (1 byte) | Argumentos... |
+------------------+------------------+-----------------------------------+
```

· Opcode: Identifica la instrucción (ej. MoveL = 0x01).

· Flags: Byte de máscara de bits. Cada bit representa un argumento opcional o un grupo. Por ejemplo:

  · Bit 0: \Conc presente

  · Bit 1: \V presente (si está en un grupo)

  · Bit 2: \T presente

  · Bit 3: \Z presente

  · Bit 4: \WObj presente

· Argumentos: Se serializan en un orden predefinido.

  · Requeridos: Siempre presentes.

  · Opcionales: Solo se escriben si su flag está activo.

  · Grupos: Solo se escribe el argumento cuyo flag está activo.

Ejemplo de serialización de MoveL (sin opcionales):

```

[0x01] [0b00000000] [datos_de_ToPoint] [datos_de_Speed] [datos_de_Zone] [datos_de_Tool]

```

Ejemplo con \V y \WObj:

```

[0x01] [0b00000010 | 0b00010000] [datos_de_ToPoint] [datos_de_Speed] [V_value (4 bytes)] [datos_de_Zone] [datos_de_Tool] [datos_de_WObj]

```

Paso 5: Estructura del Proyecto

```

rapid-serializer/

├── spec/ # Especificación en JSON

│ ├── data_types.json

│ └── instructions.json

├── generator/ # Script generador (Python)

│ └── generate.py

├── runtime/ # Librería base (en tu lenguaje objetivo)

│ ├── buffer.h/cpp # Manejo de buffer de lectura/escritura

│ └── types.h/cpp # Definiciones de tipos base (num, bool, etc.)

├── generated/ # Código generado automáticamente

│ ├── rapid_types.h/cpp # Serializadores de tipos de datos
```

```
│ └── rapid_instructions.h/cpp # Serializadores de instrucciones
└── vm/ # Tu máquina virtual que usará el serializador
    └── ...
```

Conclusión

Este enfoque te da:

· Mantenibilidad: Para añadir una nueva instrucción, solo editas el JSON y vuelves a ejecutar el generador. No tocas el código de la VM.

· Consistencia: Las reglas de serialización son las mismas para todas las instrucciones.

· Flexibilidad: Puedes generar código para cualquier lenguaje de destino cambiando las plantillas en el generador.

¿Te gustaría que profundice en alguno de estos pasos? Por ejemplo, podría mostrarte cómo sería el código Python del generador, o cómo sería la función serialize_robtarget en C/C++.

Revision #1
Created 25 March 2026 23:39:16 by Rafael Carbonell Lázaro
Updated 25 March 2026 23:41:24 by Rafael Carbonell Lázaro