Proteus 9.1 L’ ambiente di sviluppo per Linux

Proteus 9.1 è un sistema di sviluppo per la rapida creazione di applicazioni in linguaggio C/C++ eseguibili sulle apparecchiature SYEL.
Il sistema consiste in:

- una serie di librerie che coprono l’ insieme di funzioni necessarie a sviluppare un programma con particolare riguardo alla programmazione di controlli industriali.
- Un ambiente di sviluppo integrato (IDE) che gira su Windows (versioni Windows 7 e successive) con interfaccia grafica object oriented tipo Visual Basic e Delphi.
- Una serie di componenti aggiuntivi per la gestione multilingue, per la creazione di immagini in vari formati, per il debug dei programmi ecc…

Proteus può creare programmi per vari tipi di CPU e per vari sistemi operativi.
Attualmente sono supportati i sistemi:

- ARM7-DTMI per apparecchiature SYEL con sistema operativo proprietario
- ARM CORTEX M3-M4 per apparecchiature SYEL con sistema operativo proprietario
- ARM CORTEX A53-A72 con sistema operativo Linux Debian oppure Ubuntu.
- Intel X64 con sistema operativo Windows XP o successivo

Il sistema è concepito in modo che un programma sviluppato per un tipo di CPU possa girare su qualsiasi altro tipo di macchina senza modifiche, o comunque un numero minimo di adattamenti.
Questo è un vantaggio per la portabilità e per il riutilizzo del codice sviluppato, ma è anche un grosso limite per lo sviluppo di software altamente specializzato.

La versione 9.1, pur restando completamente compatibile con le versioni precedenti, per cui ogni programma sviluppato con le versioni precedenti può essere compilato con l’ attuale versione senza alcuna modifica, è stato ottimizzato per l’ ambiente Linux, per cui i programmi sviluppati per Linux possono utilizzare una serie di funzioni non presenti in altri ambienti.

Le versioni precedenti di Proteus consentono lo sviluppo di programmi scritti in puro linguaggio C, salvo lo sviluppo per controllori Pineapple, che possono usare il linguaggio C++ nell’ apposita sezione dedicata alla compilazione di programmi compatibili con Arduino.

La versione 9.1, per quanto riguarda lo sviluppo di programmi per Linux, usa la compilazione C++, pur restando completamente compatibile con il codice sviluppato per altri sistemi operativi.
In altre parole, un programma sviluppato ad esempio per Windows o per Cortex-M3 gira correttamente su Linux, ma non è detto il contrario, in quanto un programma sviluppato per Linux può contenere overload di funzioni, dichiarazione di classi e altre funzioni supportate solo dal linguaggio C++.

La versione 9.1 consente, sotto linux, di sviluppare applicazioni non grafiche e servizi, applicazioni grafiche full-screen in ambiente non grafico ed applicazioni grafiche in finestra X su sistemi con desktop Debian, Armbian, Raspbian e Ubuntu. L’ attuale versione non supporta l’ ambiente grafico Wayland.
L’ applicazione riconosce automaticamente l’ ambiente in cui sta girando.

Pagine, Task , Pipes, Timers e altre funzioni asincrone

All' avvio il programma esegue le istruzioni contenute nel file **start.c**, dopo di che il programma principale(il task 0) avvia l' interfaccia grafica attivando la pagina selezionata in start.c o, per default, la prima pagina nell' elenco delle pagine del programma.

Una pagina viene selezionata dal comando:

```
pagina = xxx;
```

dove xxx è il **numero** della pagina nell' elenco delle pagine del programma oppure il **nome** della pagina preceduto dal simbolo _ (underscore).

Esempio:

```
pagina=2;
pagina=_datifissi;
```

una pagina può uscire in due modi:

```
1. page_end = 1;
2. gopage(nome);
```

Nel primo caso il programma passa alla pagina selezionata dal comando **pagina=….**
Se non è stata selezionata alcuna altra pagina viene reinizializzata la pagina corrente.
Nel secondo caso il programma passa alla pagina selezionata con **gopage**, e se quest' ultima esce con **page_end=1** il controllo passa nuovamente alla **pagina chiamante** a partire dall' istruzione seguente al comando gopage. Questo metodo è utile ad esempio per creare delle pagine pop-up.

Una pagina è un' interfaccia grafica, che, mediante l' uso di campi di edit, bottoni, icone ed altri oggetti simili, permette l' interazione dell' utente con il programma.
A differenza di altri sistemi come ad es. Visual Basic, QT, Delphi, in cui l' interazione della pagina con l' utente avviene solo per eventi, Proteus adotta una filosofia differente:

- Ogni 50 millisecondi viene fatta una scansione completa di tutti gli oggetti della pagina.
- Ogni oggetto è associato ad una **variabile**.
- Se la variabile cambia di valore l' oggetto viene aggiornato di conseguenza.
- Se un oggetto viene selezionato con un click del mouse, o con il touch screen, la sua variabile associata cambia di valore di conseguenza.

Una pagina è un oggetto lento, con una tempistica adatta ai tempi di reazione dell' utente, quindi, per quanto possibile, è **sconsigliato di usare nella pagina funzioni di controllo** dei dispositivi collegati all' apparecchiatura (ingressi, uscite, allarmi …). Questo compito è generalmente demandato ad altri componenti del programma (plc, task, routines a tempo, routines di callback, interrupts).
Tutte le pagine hanno la stessa dimensione, specificata nel project manager, pertanto se vogliamo creare delle pagine di pop-up con dimensioni ridotte possiamo farlo impostando il fondo della pagina trasparente e creando un pannello delle dimensioni volute includendovi gli altri oggetti della pagina.
Il limite attualmente fissato per il massimo numero di oggetti in una pagina è di **512**, mentre non c' è un limite per il numero di pagine di un progetto.

Gestione dei task in Proteus 9.1 sotto Linux

Un task è un thread di Linux, cioè un flusso di istruzioni che girano indipendentemente dal task principale e dagli altri task. In un controllore multi-cpu più task possono girare contemporaneamente su cpu diverse, ed in ogni caso esiste un meccanismo di reskeduling preemptive che distribuisce il tempo macchina tra i vari threads. Per evitare che un task usi più tempo macchina del necessario rallentando inutilmente gli altri è bene, dove possibile, usare alcuni accorgimenti:

- Usare **meccanismi di tipo blocking** nell' attesa di flussi di dati esterni (COM, Ethernet …)
- Nel caso di polling usare dei **wait()** o degli **idle()** nei loop di attesa.

Proteus usa **due** tipi diversi di task:
Task normali, lanciati dal comando
`exec_task(…);`
Task hidden lanciati dal comando:
`exec_task_hidden(…);`

Entrambi i tipi di thread vengono lanciati mediante il comando pthread_create(…).
Le differenze sono comunque importanti:

task normale:

Si possono eseguire contemporaneamente fino ad un massimo di 32 task normali.
I comandi per gestire tale tipo di task sono:

int exec_task(void (*sub)(void),unsigned int num=0,void *para=0,int prior=0)

sub è il nome del task.
Gli altri parametri (facoltativi) sono:
num = numero del task. È sconsigliato usarlo in quanto se c' è già un task in esecuzione con lo stesso numero il task non andrà in esecuzione. È comunque utile nel caso in cui sia lanciata la stessa routine in più task, in quanto è l' unica maniera di distinguerli. Se num è uguale a zero il task viene allocato nel primo slot libero dei 32 disponibili. Il valore di ritorno è il numero dello slot in cui è stato allocato, oppure zero se tutti gli slot sono occupati.
para è il puntatore ad una eventuale lista di parametri da passare al task.
prior è la priorità del task. Può essere zero (priorità normale) oppure un numero compreso tra 1 e 99, nel qual caso il task è eseguito in modalità SCHED_FIFO con la priorità indicata. Una priorità troppo alta (>40) può bloccare o comunque rallentare la ricezione dei dati da Ethernet per il task stesso e il regolare funzionamento degli altri threads di Linux.
Un task normale ha una **propria struttura dati** e può accedere alle funzioni grafiche conservando i propri puntatori , i propri font e i propri colori indipendentemente dagli altri task.
Può inoltre essere gestito mediante le seguenti funzioni in cui il parametro task è il nome oppure il numero del task:

int suspend_task(unsigned int task);

sospende il task

int resume_task(unsigned int task);

riavvia l' esecuzione di un task sospeso

int remove_task(unsigned int task);

elimina il task

un task può conoscere il proprio numero di thread chiamando la funzione di Linux:

```
pthread_t pthread_self(void);
```

e può conoscere il numero di slot a cui è stato assegnato chiamando la funzine:

```
int NTask(pthread_t TASK);
```

con argomento `pthread_self`

o più semplicemente usando la macro equivalente

NTASK

Esempio:

```
int mythread = pthread_t pthread_self();
int myslot = NTask(mythread);
```

oppure:

```
int myslot = NTASK;
```

Mentre conoscere il numero del proprio thread non serve a niente, la conoscenza del numero dello slot può essere molto utile nell' implementazione di routine a **codice rientrante**:

Una routine a codice rientrante può essere chiamata contemporaneamente da più task senza generare interferenze.
A tale scopo la routine non deve usare variabili globali né variabili statiche (ed ovviamente non deve chiamare al suo interno altre routines non rientranti).
L' uso di sole variabili locali può gravare pesantemente sulla dimensione dello stack, per cui un altro metodo è quello di usare variabili globali o statiche vettorializzate, usando come indice del vettore il numero di slot del task chiamante.

Esempio:

caso 1: la routine usa solo variabili locali (allocate nello stack) … ma in questo caso dopo il return non è detto che il vettore puntato come risultato sia ancora valido, quindi …**ERRORE,** non si può fare così.

```
char *stima(int n)
{
   char vett[10];
   if (n < 10) sprintf(vett,"basso");
   else sprintf(vett,"alto");
   return vett;
}
```

Caso 2: la routine usa variabili statiche (allocate in ram) per cui all' uscita i valori sono conservati, ma **non è rientrante** in quanto se chiamata contemporaneamente da più task può sovrascrivere il risultato.

```
char *stima(int n)
{
   static char vett [10];
   if (n < 10) sprintf(vett,"basso");
   else sprintf(vett,"alto");
   return vett;
}
```

In questo caso comunque la routine può essere chiamata da più task contemporaneamente usando dei **semafori di mutua esclusione**.(i semafori da 1 a 29 sono usati internamente dal sistema operativo)

Task a:

```
_p(30); printf("%s\n",stima(2));_v(30);
```

Task b:

```
_p(30); printf("%s\n",stima(12));_v(30);
```

Caso 3: la routine usa variabili statiche vettorializzate allocate in ram, per cui all' uscita i valori sono conservati e la routine è **rientrante**.

```
char *stima(int n)
{
   static char vett[MAX_TASK_NUMBER][10];
   if (n < 10) sprintf(vett[NTASK],"basso");
   else sprintf(vett[NTASK],"alto");
   return vett[NTASK];
}
```

i task normali sono gli unici che possono utilizzare le seguenti funzioni rientranti:

```
int  every(unsigned long t);
void everyUs(unsigned long t);
```

che servono per sospendere il task fino alla scadenza del tempo indicato nell' argomento t rispettivamente in millisecondi e in microsecondi.
A differenza del comando wait() le suddette funzioni consentono di sospendere e riprendere l' esecuzione ad intervalli di tempo regolari, indipendentemente dal tempo impiegato dalle istruzioni stesse:

Esempio:
il comando printf viene eseguito **esattamente** ogni 2 millisecondi.
Il jitter è dell' ordine di pochi microsecondi.

```
while(1)
{
   every(2);
   printf(….);
}
```

il comando printf viene eseguito ogni 2 millisecondi **più la durata del comando printf()**.

```
while(1)
{
   wait(2);
   printf(….);
}
```

Se la durata della routine è maggiore del tempo specificato in every il tempo perduto **non viene recuperato** e il programma riprende al successivo multiplo del periodo indicato in every, così ad esempio se il printf() dovesse durare 3 millisecondi verrà eseguito dopo 4 millisecondi anziché 2 millisecondi.
Questo è necessario per evitare che, se ad esempio il task viene sospeso per 1 secondo, alla riattivazione esegua a raffica 500 printf() prima di rimettersi in pari.

task hidden:

Il limite di task hidden che possono essere lanciati contemporaneamente è di 1000.
È preferibile usare task hidden dove non è necessario usare la grafica o altre routines rientranti, per non gravare sul numero abbastanza esigui di task normali. Proteus usa decine di task hidden per gestire le risorse di sistema quali Ethernet, le COM, il mouse, la tastiera, il CAN, i timers ecc…

I comandi relativi ai task hidden sono:

unsigned int exec_task_hidden(void (*sub)(void),unsigned int arg=0,int prior=0);

sub è il nome della routine da eseguire come task.
arg (facoltativo) è un parametro da passare alla routine.
prior è la priorità (come per i task normali)
il valore di ritorno è il numero del thread del task. Serve se si vuole rimuoverlo.

int remove_task_hidden(unsigned int num);

rimuove il thread con il numero specificato.

Al contrario di altri sistemi operativi in Linux un thread può terminare con un return.
Perciò una stessa routine può essere lanciata indifferentemente come task (con exec_task(..) o exec_task_hidden(…)) o come subroutine. L' unica differenza è che nel primo caso l' esecuzione della routine procede in parallelo con il task chiamante, mentre nel secondo caso il task chiamante continua solo dopo la fine della routine.

Importante ricordare che i task possono scambiarsi i dati mediante **variabili globali**, che in tal caso debbono essere dichiarate **con il prefisso volatile** affinché la cosa funzioni correttamente.

In proteus per default è abilitato un secondo task , il cui source può essere scritto nel file **plc.c.**
Se non vogliamo attivarlo, usando il comando

```
#define NO_PLC
```

possiamo disabilitare tale task. Alcune funzioni utilizzate per gestire gli ingressi di un PLC funzionano regolarmente **solo se il task plc non è disattivato**. Tali funzioni sono:

```
Pulseon
Pulseoff
Delay
DelayUp
DelayDown
MonostableUp
MonostableDown
```

Qualora volessimo usare queste funzioni in un task diverso dal plc se il task plc è disattivato, occorre includere all' inizio del loop principale di tale task l' istruzione:

```
start_cycle();
```

In quest' ultimo caso però nonè detto che la cosa funzioni se ci si trova in un loop secondario che non passa dalla suddetta istruzione, per cui **se si vogliono utilizzare tali funzioni** è bene **non disabilitare il task plc**.

forks e pipes

Un programma può lanciare comandi della shell di Linux mediante le funzioni sotto descritte.

funzioni principali:

void ExecWait(const char *format, ...);

void ExecNoWait(const char *format, ...);

La sintassi è la stessa del comando printf(…) e l' argomento è una stringa formattata.
L' unica differenza tra i due comandi è che il primo lancia un processo creando un fork ed eseguendo il comando contenuto nella stringa formattata invocando la shell di Linux ,attendendo la sua conclusione prima di proseguire, mentre il secondo comando prosegue senza attendere la conclusione del comando lanciato. A differenza di popen(…) non si possono passare comandi né ricevere risposte dal processo.

Esempio:
il comando termina i processi di Linux myprocess3, myprocess4 e mytprocess5

```
for (int i=3;i<6;i++)
ExecNoWait("sudo pkill myprocess%d",i);
```

FILE *popen(const char **command*, const char **type*);

Questa funzione apre un processo creando una pipe e fa un fork invocando la shell di Linux.
La pipe è monodirezionale ed Il tipo può essere "r" se la pipe è di input o "w" se si vuole creare una pipe di output. Ritorna con il numero del file della pipe, che verrà usato per chiuderla con il comando pclose.
Si può comunicare con la pipe tramite tutte le funzioni standard di input e di output per i files.

int pclose(FILE **stream*);

La funzione attende che il comando lanciato con popen sia terminato, quindi chiude la pipe e continua l' esecuzione del programma.

Esempio:
questa routine legge il nome di una eventuale chiavetta usb inserita usando il comando di shell blkid.
Il comando blkid dà una risposta del genere:

```
pi> sudo blkid /dev/sda1
/dev/sda1: SEC_TYPE="msdos" LABEL="SYEL1" UUID="FAA1-00F2" TYPE="vfat" PARTUUID="10c598e1-01"
```

```
char *find_usb_label(void)
{
    static char keyname[80];                //static per conservare il contenuto dopo il return
    char TempData[80];                      //quindi la routine non è rientrante
    FILE *fp;                               //la pipe
    fp = popen("sudo blkid /dev/sda1", "r"); //chiede info sul device sda1 (chiavetta usb)
    fgets(TempData, 80, fp);                //le legge
    fclose(fp);                             //chiude la pipe
    char *s=strstr(TempData ,"LABEL=");     //cerca il parametro che ci interessa
    if (s==0) return 0;                     //se non trovato esce
    int n = sscanf(s,"LABEL=%s",keyname);   //scansiona per estrarre il nome della chiavetta
    if (n)                                  //se lo trova toglie gli apici dal nome della chiavetta
    {
       keyname[strlen(keyname)-1]=0;        //toglie l' apice finale …
       return (&keyname[1]);                //e quello iniziale
    }
    else return 0;                          //chiavetta non inserita o senza nome
}
```

timers e altre funzioni temporali

reset_timer(void)
Azzera il tempo assoluto.

U32 _timer(void);
Ritorna con il tempo assoluto in millisecondi trascorso dall' accensione della macchina o dall' ultimo reset.

double ftime(void);
Ritorna con il tempo assoluto in millisecondi e frazioni di millisecondo (in floating point) trascorso dall' accensione della macchina o dall' ultimo reset.

U32 micro_timer(void);
Ritorna con il tempo assoluto in microsecondi trascorso dall' accensione della macchina o dall' ultimo reset.

U32 nano_timer(void);
Ritorna con il tempo assoluto in nanosecondi trascorso dall' accensione della macchina o dall' ultimo reset.

void wait (U32 delayInMs);
Sospende il task per il tempo impostato in millisecondi.

void delayMs(U32 delayInMs);
Alias di wait (): Sospende il task per il tempo impostato in millisecondi.

void delayUs(U32 delayInUs);
Sospende il task per il tempo impostato in microsecondi

void delayNs(U32 delayInNs);
Sospende il task per il tempo impostato in nanosecondi.

int every(unsigned long t);
Come visto sopra, se inserito in un loop consente l' esecuzione del loop a cadenza regolare di t millisecondi.

void everyUs(unsigned long t);
Se inserito in un loop consente l' esecuzione del loop a cadenza regolare di t microsecondi.

int exec_timer(void(*subroutine)(),time_t interval,time_t fra_quanto);
Un timer è una funzione che può essere richiamata ad intervalli di tempo regolari. In Linux una funzione può essere chiamata indifferentemente (e anche contemporaneamente) come subroutine, come task o come timer. Solo nel primo caso interrompe il flusso del programma chiamante, mentre negli altri casi viene eseguita in un thread diverso in parallelo al task chiamante.

Tuttelle funzioni eseguite come timers vengono eseguite nello stesso thread, e se il tempo scade contemporaneamente per più funzioni, l' ordine di esecuzione è quello in cui sono state attivate.
Si possono attivare al **massimo 100 timers** contemporaneamente.

I parametri sono:
Subroutine: è il nome della funzione
Interval: è la cadenza con cui la funzione deve essere eseguita, espressa in millisecondi.
Fra_quanto: è il tempo a partire dall' attivazione che deve trascorrere prima della prima esecuzione.
Se si esegue **un nuovo exec_timer** per la stessa funzione, il tempo della prima esecuzione viene **azzerato**. In questo modo è possibile utilizzare questa funzione come **watchdog**, lanciandola a tempi regolari come timer con un tempo di prima esecuzione maggiore della cadenza con cui viene richiamata, in modo che in condizioni normali non venga mai eseguita.

Esempio:
```
exec_timer(my_function,10,50);
```
la funzione viene eseguita ogni 10 millisecondi, per la prima volta fra 50 millisecondi.

```
exec_timer(my_function,ONCE,50);
```
la funzione viene eseguita una sola volta, fra 50 millisecondi.(è una call differita)

```
exec_timer(my_function,10,NOW);
```
la funzione viene eseguita ogni 10 millisecondi, per la prima volta immediatamente.

```
exec_timer(my_function,ONCE,NOW);
```
la funzione viene eseguita una sola volta, immediatamente. È del tutto equivalente ad un
```
exec_task_hidden(my_function);
```

i due ultimi parametri possono avere i suffissi:

SEC	**secondi**
MIN	**minuti**
HOUR HOURS	**ore**
DAY DAYS	**giorni**
WEEK WEEKS	**settimane**

Esempio:
```
exec_timer(my_function,2 DAYS,3 WEEKS + 1 DAY + 3 HOURS + 5 MIN + 18 SEC);
```
la funzione sarà eseguita ogni 2 giorni a partire da: tra 22 giorni, 3 ore, 5 minuti e 18 secondi.
Il **massimo intervallo di tempo** dichiarabile è di 2^31 millisecondi ovvero 24 giorni, 20 ore e 31 minuti.

int suspend_timer(void(*subroutine)());

sospende la funzione fino alla eventuale riattivazione.

int resume_timer(void(*subroutine)());

riattiva una funzione sospesa.

int sleep_timer(void(*subroutine)(),time_t tempo);

sospende una funzione per il tempo indicato.

Esempio:

```
exec_timer(my_function,10,50);
……
Sleep_timer(my_function,10 SEC);
……
```

la funzione viene eseguita ogni 10 millisecondi, per la prima volta fra 50 millisecondi.
Ad un certo momento viene sospesa per 10 secondi.
La funzine riprende dopo 10 secondi oppure all' occorrenza di una nuova exec_timer.

int remove_timer(void(*subroutine)());

Rimuove il timer lasciando lo slot libero per eventuali altri timers.

Funzioni riguardanti il realtime clock

void set_time(char *s);

setta l' ora attuale.
La stringa s è del tipo "hh:mm.ss"
Se la macchina ha accesso a Internet o ha un realtime clock questa funzione è inutile.

void set_date(char *s);

setta la data attuale.
La stringa s è del tipo "gg/mm/aa"
Se la macchina ha accesso a Internet o ha un realtime clock questa funzione è inutile.

char *get_time(void);

richiede l' ora attuale.
La stringa di ritorno della funzione è del tipo "hh:mm.ss"

char *get_date(void);

richiede la data attuale.
La stringa di ritorno della funzione è del tipo "gg/mm/aa"