pyplc

Выпуск v0.1.9

Эталон КОМ

мая 07, 2024

Состав:

Глава 1

Термины и обозначения

ПЛК - программируемый логический контроллер.

Логика управления - программа для специализированного оборудования (программируемые контроллеры), которая выполняет управление промышленным оборудованием.

SCADA (аббр. от англ. Supervisory Control And Data Acquisition — диспетчерское управление и сбор данных) — программный пакет, предназначенный для разработки или обеспечения работы в реальном времени систем сбора, обработки, отображения и архивирования информации об объекте мониторинга или управления.

callback (англ. call — вызов, англ. back — обратный) или функция обратного вызова в программировании — передача исполняемого кода в качестве одного из параметров другому коду. Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове. Этот код может быть определён в других контекстах программного кода и быть недоступным для прямого вызова из этой функции. Некоторые алгоритмические задачи в качестве своих входных данных имеют не только числа или объекты, но и действия (алгоритмы), которые естественным образом задаются как обратные вызовы. (страница на wikipedia)

Глава 2

Общая информация

PYPLC - библиотека для разработки и отладки логики управления на языке программирования micropython/python для контроллеров KRAX PLC-932. Цель библиотеки - сделать программирование контроллера на python схожим с написанием на языках программирования IEC-61131 (LD,SFC,STL), обеспечить связь с программой визуализации (интерфейс оператора, SCADA).

Логика управления из себя представляет циклическую программу, каждый цикл которой состоит из:

- опрос модулей ввода для получения состояния измерительных каналов (аналоговых, дискретных, счетчиков импульсов)
- обработка состояний измерительных каналов и, возможно, формирование управляющих воздействий
- отправка управляющих воздействий модулям вывода (аналоговые, дискретные)

Глава 3

Разработка логики

Чтобы сделать программирование ПЛК на python похожим на IEC-61131, используется три подхода:

- STL - функции или потомки *POU*
- SFC - генераторы или потомки *SFC*
- LD - экземпляры классов из `pyplc.ld`.

3.1 Работа с параметрами

Важный аспект в процессе написания программы - это передача параметров, измерительных и управляющих каналов логике и обратно. При разработке если функции необходимо передать изменяющийся параметр (например состояние датчика), то используется механизм callback функций, то-есть вместо значения состояния датчика передавать надо функцию, которая если ее вызвать возвращает текущее состояние.

По этой причине классы из PYPLC переопределяют специальный метод $\_\_call\_\_$, то позволяет использовать их экземпляры как callback функции.

3.2 Простая программа

Чтобы программа была похожа на STL реализуем логику как обычную функцию или как потомка *POU*.

С использованием PYPLC программа выглядит так:

```
from pyplc.config import plc

def user_prog():
   pass

plc.run(instances=[user_prog],ctx=globals())
```

Логика здесь ничего не делает (*user_prog* содержит пустую инструкцию *pass*). Она будет выполняться периодически (по умолчанию каждые 100 мсек). Опрос и организацию переодического запуска выполняет последняя строка, *run()*. Выход из программы возможен в случае KeyboardInterrupt (Ctrl+C) либо по ошибке в коде.

С применением *POU* нужно наследовать новый класс от него, переопределить метод `__call__()`

```
from pyplc.config import plc
from pyplc.pou import POU

class UserProg(POU):
   def __call__(self):
      pass

user_prog = UserProg()

plc.run(instances=[user_prog],ctx=globals())
```

Классы с методом *__call__* могут использоваться как функции: *user_prog()* произведет вызов метода *__call__* нашего класса UserProg.

3.3 Пошаговые программы

Чтобы программа была похожа на SFC нужно написать функцию-генератор или потомок *SFC*.

Удобно использовать для того чтобы написать программу, которая выполняет различные последовательности действий каждый следующий цикл сканирования.

```
from pyplc.config import plc

def user_prog():
   print('тик')
   yield
   print('так')
   yield

plc.run(instances=[user_prog],ctx=globals())
```

Генератор, если сильно упрощенно - это функция, в теле которой есть ключевое слово *yield*. Если plc.run в параметре instances встречается генератор, то будет произведен его обход (возвращаемое значение yield игнорируется), по одному элементу за цикл, или проще говоря *user_prog* каждый цикл будет выполнена до следующего *yield*. В примере выше получится в первый цикл работы user_prog напишет „тик“, во второй цикл работы напишет „так“, затем снова „тик“ и так без конца (когда генератор кончается, он перезапускается сначала).

С применением *SFC* нужно наследовать новый класс от него, переопределить метод *main()*

```
from pyplc.config import plc
from pyplc.sfc import SFC

class UserProg(SFC):
   def main(self):
      print('тик')
      yield
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      print('так')
      yield

user_prog = UserProg()

plc.run(instances=[user_prog],ctx=globals())
```

3.4 Многоступенчатые программы

Чтобы программа была похожа на LD необходимо использовать методы `pyplc.ld.LD`. Их всего 2: *no* или *nc*. Им соответствуют нормально открытые и нормально-закрытые контакты (contact). Эти методы возвращают объект `Cell`, которые в свою очередь имеют методы для создания контактов (contact), катушек (coil) и набор других объектов LD логики.

```
from pyplc.config import plc
from pyplc.channel import IBool,QBool
from pyplc.ld import LD

SWITCH_ON_1 = IBool.at( '%IX8.0' )
SWITCH_OFF_1= IBool.at( '%IX8.1' )
POWER_ON_1 = QBool.at( '%QX9.0' )

user_prog = LD.no(SWITCH_ON_1).set(POWER_ON_1).end()

plc.run(instances=[user_prog],ctx=globals())
```

Логика user_prog похожа на цепочку, начинается с *LD* заканчивается *end()*. Результат тоже может вызываться как функция и может быть передана в параметр *instances* *run()*

3.5 Отладка логики

В процессе разработки программы с использованием PYPLC отладка проводится в 2 этапа:

- Первичная, на компьютере.
- На контроллере.

На первом этапе доступны привычные отладочные механизмы: контрольные точки, просмотр значений всех переменных и памяти. Ничем не отличается от отладки обычной программы на python, используется ваша любимая среда разработки (я поклонник vscode). Используя *force()* можно написать логику создания имитационных значений, которую в конечном варианте логики не вызывать или закомментировать. Также можно настроить режим симуляции (когда логика выполняется на компьютере, но каналы ввода-вывода обновляются на контроллере).

На втором этапе, в контроллере, доступны только просмотр значений переменных и изменение их значений. Происходит это в режиме командной строки, из telnet клиента. Реализовано это при помощи *CLI*

3.5.1 Сервер командной строки, порт 2455

Пример использования:

```
from pyplc.utils.cli import CLI
telnet = CLI()
while True:
    telnet()
```

Каждый цикл работы программы читает данные от клиента и выполняет их через eval(). Специально создавать экземпляр данной программы не нужно, это происходит в *pyplc.config*.

Пример подключения (предполагаем контроллер имеет IP 192.168.1.120)

```
$ telnet 192.168.1.120 2455
Trying 192.168.1.120...
Connected to 192.168.1.120.
Escape character is '^]'.
>>>
```

class pyplc.utils.cli.CLI(*port=2455*)

Поддержка telnet протокола на порту 2455. Полученные данные обрабатываются при вызове данной программы.

Глава 4

Работа с физическими сигналами

В PYPLC реализовано несколько типов сигналов, которые можно подключить к контроллеру для получения состояния и передачи управляющих воздействий:

- Дискретные: *QBool* , *IBool*
- Аналоговые: *IWord* , *QWord* , *ICounter8*

Объявить каналы можно явно в коде или в файле krax.csv. Классы, связанные с каналами ввода-вывода объявлены в модуле *pyplc.channel*

С контроллером связывается область памяти ввода-вывода фиксированного размера, каждый канал прикреплен к участку в этой памяти. Дискретные каналы прикреплены к одному биту, аналоговые к одному или нескольким байтам.

4.1 Объявление каналов в коде

Настройку каналов можно производить явно в коде до вызова *run()* :

```
MOTOR_ISON = IBool.at('%IX0.0')
```

4.2 Объявление каналов в файле

Каналы ввода-вывода можно объявить с помощью файла krax.csv. Это обычный файл, в каждой строке которого объявляется один канал (при этом используется содержимое файла krax.json). Формат строк krax.csv (кроме первой, которая выполняет роль заголовка) должен быть следующим:

```
<имя>;<тип>;<модуль>;<канал>
```

например:

```
MIXER_ISON;DI;1;1
MIXER_ROT;CNT8;1;9
MIXER_ON;DO;2;1
MIXER_I;AI;3;1
MIXER_FQ;AO;4;1
```

Номер модуля N используется при вычислении адреса в памяти ввода-вывода. Для этого из файла krax.json вычисляется сумма элементов slots[0..N-1]. Номер канала это либо номер бита (для DI/DO), либо номер слова (AI/AO), либо смещение байта+8 для CNT8 (каналы 1-8 используются как обычные DI)

Пример файла krax.json

```
{
   "slots": [2,1,8,8],
   "node_id": 1,
   "init": {
      "flags": 0,
      "iface": 0,
      "hostname": "plc",
      "rate": 12
   },
   "layout": [],
   "devs": [],
   "via": "0.0.0.0",
   "scanTime":100
}
```

4.3 Модуль pyplc.channel

Базовый класс для измерительных и управляющих каналов *Channel*. В нем реализован механизм подписи на изменения значения (*bind()*), запись/чтение текущего значения и метод для синхронизации с памятью ввода-вывода (`sync()`).

Потомки Channel могут использоваться как функция с одним параметром или без параметров (метод *__call__()*). Если вызвать без параметров, то она вернет значение канала, а если с одним параметром, то будет выполнена попытка записать значение.

```
if not MOTOR_ISON(): #може что MOTOR_ISON.read()
   MOTOR_ON(True) #може что MOTOR_ON.write(True)
```

Классы

Channel([name, init_val, rw])	Основа для всех измерительных каналов
IBool(addr, num[, name])	Дискретный вход (логический True/False)
QBool(addr, num[, name])	Дискретный выход (логический True/False)
ICounter8(addr[, name])	Вход-счетчик импульсов 8-битный
IWord(addr[, name])	Аналоговый вход 16 бит
QWord(addr[, name])	Аналоговый выход 16 бит

4.3.1 class Channel

class pyplc.channel.Channel(*name=''*, *init_val=None*, *rw=False*)

Основа для всех измерительных каналов

Параметры

- name (*str , optional*) – Имя измерительного канала. По умолчанию „“.
- init_val (*bool | int , optional*) – Начальное значение. По умолчанию None.
- rw (*bool , optional*) – Тип измерительного канала (можно изменять или только читать). По умолчанию только читать.

__call__(*value=None*)

Доступ к значению для чтения/записи.

Параметры
value (*Any , optional*) – если value!=None, то производится *write()*

Результат
Если value==None возвращает результат вызова *read()*

Тип результата
bool|int

bind(*callback*)

Соединить канал IO с функцией оповещения. Если значение канала изменится, то будет вызвана функция callback. При этом возвращается функция, с помощью которой можно производить запись в IO, если это доступно. Отменить вызов callback можно *unbind()*

Параметры
callback (*function*) – что вызвать при изменении канала

Результат
функция для доступа к изменению переменной

Тип результата
callable

force(*value*)

Обеспечивает маханизм записи имитационных значений

Измерительные каналы с режимом только чтение при разработке удобно имитировать как будто измерено новое значение, например сигнал MOTOR_ISON можно изменить программно если выдан сигнал MOTOR_ON.

Параметры
value (*Any*) – имитируемое значение

read()

Получить значение измерительного/управляющего канала

Результат
текущее значение

Тип результата
bool|int

unbind(*callback*)

Убрать оповещение указанного callback

Параметры
callback (*function*) – Может быть функцией, а также результатом id(<функция>), где <функция> ранее использовалась в bind

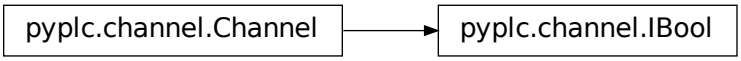
`write(value)`

Изменить текущее состояние канала

Если новое значение отличается от старого, то произойдет оповещение всех функций, переданных *Channel.bind()*

Параметры
value (*bool* | *int*) – новое значение

4.3.2 class IBool



Из программы

```
MIXER_ISON = IBool.at('%IX0.1')
```

Строка адреса должна начинаться с %IX, далее идет адрес байта (0), затем после точки номер бита (1).

`class pyplc.channel.IBool(addr: int, num: int, name='')`

Дискретный вход (логический True/False)

Параметры

- addr (*int*) – номер байта (смещение) в памяти ввода-вывода
- num (*int*) – номер бита
- name (*str, optional*) – имя канала ввода-вывода.

`static at(addr: str) → IBool`

Создать канал и закрепить его за указанным адресом

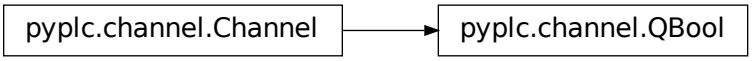
Параметры
addr (*str*) – Адрес, должен начинаться с %IX, затем смещение, «.» и номер бита, например %IX0.1

Исключение
RuntimeError – Формат адреса не соответсвует требованиям

Результат
Новый канал

Тип результата
IBool

4.3.3 class QBool



Из программы

```
MIXER_ON = QBool.at('%QX0.1')
```

Строка адреса должна начинаться с %QX, далее идет адрес байта (0), затем после точки номер бита (1).

class pyplc.channel.QBool(*addr*, *num: int*, *name=''*)

Дискретный выход (логический True/False)

Параметры

- addr (*int*) – адрес байта
- num (*int*) – номер бита
- name (*str, optional*) – имя канала.

static at(*addr: str*) → *QBool*

Создать канал и закрепить его за указанным адресом

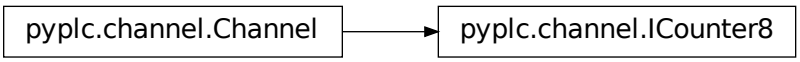
Параметры
addr (*str*) – Адрес, должен начинаться с %QX, затем смещение, «.» и номер бита, например %QX0.1

Исключение
RuntimeError – Формат адреса не соответсвует требованиям

Результат
Новый канал

Тип результата
QBool

4.3.4 class ICounter8



Из программы

```
MIXER_ROT = ICounter8.at('%IB8')
```

class pyplc.channel.ICounter8(*addr*, *name=''*)

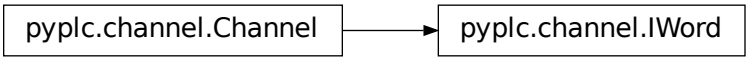
Вход-счетчик импульсов 8-битный

В памяти ввода-вывода занимает 1 байт. Значение может принимать как обычный int. Переполнение байта учитывается программно когда байт из памяти ввода-вывода изменяется в меньшую сторону. Например: значение 253 - 254 - 255 (переполнение) - 5, значение канала будет 300.

Параметры

- addr (*int*) – адрес байта
- name (*str, optional*) – имя канала

4.3.5 class IWord



Из программы

```
MIXER_I = IWord.at('%IW3') #или IWord.at('%IB6')
```

class pyplc.channel.IWord(*addr*, *name=''*)

Аналоговый вход 16 бит

Параметры

- addr (*int*) – адрес первого байта
- name (*str, optional*) – имя канала

static at(*addr: str*) → *IWord*

Создать канал и закрепить его за указанным адресом

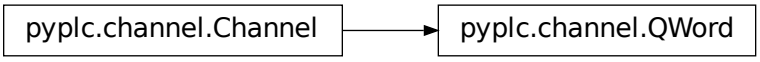
Параметры
addr (*str*) – Адрес, должен начинаться с %IW или %IB, затем смещение (в 16-битных словах или байтах соответственно)

Исключение
RuntimeError – Формат адреса не соответсвует требованиям

Результат
Новый канал

Тип результата
IWord

4.3.6 class QWord



Из программы

```
MIXER_FQ = QWord.at('%QW3') #или QWord.at('%QB6')
```

class pyplc.channel.QWord(*addr*, *name=''*)

Аналоговый выход 16 бит

Args: addr (int): адрес первого байта name (str, optional): имя канала

static at(*addr: str*) → *QWord*

Создать канал и закрепить его за указанным адресом

Параметры
addr (*str*) – Адрес, должен начинаться с %QW или %QB, затем смещение (в 16-битных словах или байтах соответственно)

Исключение
RuntimeError – Формат адреса не соответсвует требованиям

Результат
Новый канал

Тип результата
QWord

Глава 5

API

5.1 class PYPLC

Класс для организации циклической работы пользовательской логики.

- Организует цикл, в котором происходит опрос модулей ввода-вывода, затем поочередный вызов по списку пользовательских программ.
- Производит обновление измерительных каналов (переменных ввода-вывода)
- Обеспечивает обмен по TCP пользовательскими свойствами *POU*, используется при создании визуализации (SCADA)
- Обеспечивает отладочный интерфейс по telnet-подобному протоколу (порт 2455). см. *Использование Telnet*
- Используется для режима симуляции, см. *Отладка в режиме симуляции*

Программа логики контроллера должна содержать один экземпляр этого класса. Как правило создается в служебном модуле *pyplc.config* (но можно и явно самостоятельно создать).

`class pyplc.core.PYPLC(`*io_size: int*, *krax=None*, *pre=None*, *post=None*, *period: int = 100*`)`

Реализация управления циклом работы программы.

Вызывается обычно из pyplc.config. pre/post настраиваются так, чтобы к программе можно было подключиться с помощью telnet (диагностика, отладка) и с помощью pyplc.utils.subscriber для реализации интерфейса оператора на ПК.

Параметры

- `io_size` (*int*) – Размер доступной памяти ввода-вывода. Должно быть <200 байт
- `krax` (*модуль*, *optional*) – Объект, который производит синхронизацию памяти ввода-вывода.
- `pre` (*list[]*, *optional*) – Список функций, которые надо вызвать перед пользовательскими программами. Defaults to None.

- post (*_type_ , optional*) – Список функций, которые надо вызвать после пользовательских программ. Defaults to None.
- period (*int , optional*) – Период работы . Defaults to 100 (мсек).

run(*instances=None*, ***kwds*)

Запуск работы пользовательских программ.

Именованные параметры будут переданы в config.

Параметры

instances (*callable | generator , optional*) – Пользовательские программы.

5.1.1 Модуль pyplc.config

Инициализация библиотеки PYPLC, экземпляра *PYPLC* и каналов ввода вывода

```
from pyplc.config import board,plc,hw
```

- board - Экземпляр Board() для доступа к светодиодам и переключателям, также к EEPROM
- plc - Экземпляр PYPLC() для организации опроса I/O и циклической работы
- hw - Экземпляр PYPLC.State для доступа к значениям переменных I/O

Если выполняется не на контроллере, то вместо модуля config будет использован coupler, и будет использован режим имитации (simulator). Это позволяет не менять программу, которая может выполняться на компьютере в среде python.

При загрузке этого модуля используются файлы krax.json (информация о настройках опроса модулей) и krax.csv (настройка переменных ввода-вывода).

5.2 class POU

Базовый класс для всех программ (логик) с использованием PYPLC. Главная задача этого класса заключается в организации механизма обмена параметрами и измерительными каналами на основе callback функций, обеспечение подключения к входным параметрам выходов других программ.

Для использования необходимо унаследовать его в вашем классе и переопределить специальный метод *__call__*.

Чтобы в программе потомке *POU* появился новый входной параметр *clk:bool* и выход *q:bool* необходимо сделать следующее:

```
class UserProg(POU):
   clk = POU.input(False)  #объявили вход, начальное значение False
   q   = POU.output(False) #объявили выход, начальное значение False
   def __init__(self,clk:bool=False,q:bool=False, id:str = None,parent:POU = None):
      super().__init__( id, parent )
      self.clk = clk
      self.q = q

   # далее остальная часть класса UserProg
   ...
```

Специальная функция __init__ (конструктор) в качестве параметров имеет *clk: bool*, *q:bool*, но реализация *input* и *output* позволяет в качестве параметра *clk*, *q* передавать функцию, которая возвращает *bool* (clk) и принимает может принимать параметр. В теле методов *UserProg* обращение к *self.clk* будет аналогично вызову этих функции.

```
def callback_true():
   return True

user_prog = UserProg(clk = callback_true, q = print )
```

В примере выше обращение к свойству *user_prog.clk* будет эквивалентно вызову *callback_true()*, а *user_prog.q = False* эквивалентно *print(False)*.

Если необходимо выход одной программы подключить к входу другой, то используем свойства класса для получения callback функций доступа

```
user_prog1 = UserProg(q = print )
user_prog2 = UserProg(clk = UserProg.q(user_prog1), q = UserProg.clk(user_prog1))
```

UserProg.q(user_prog1) возвращает функцию f(x), которая определена как *user_prog1.q = x*.

`class pyplc.pou.POU(`*id: str | None = None*, *parent:* POU *| None = None*`)`

`NOW = 0`

момент начала цикла работы логики в нано-сек

`NOW_MS = 0`

момент начала цикла работы логики в мсек

`export(`*_POU__name: str*, *initial=None*`)`

Во время выполнения создает новый атрибут с функцией как POU.var

Параметры

- `__name` (*str*) – имя атрибута
- `initial` (*_type_ , optional*) – начальное значение

`class input(`*init_val*, *hidden: bool = False*, *persistent: bool = False*`)`

`class output(`*init_val*, *hidden: bool = False*, *persistent: bool = False*`)`

`class var(`*init_val*, *hidden: bool = False*, *persistent: bool = False*, *notify: bool = True*, *dynamic: bool = False*`)`

Локальная переменная POU. Доступна для подключения извне по протоколу TCP

5.3 class SFC

Базовый класс для написания пошаговой логики. Для использования необходимо унаследовать ваш класс от SFC и переопределить метод *pyplc.sfc.SFC.main()*. Далее в коде на месте, где должен быть переход на следующий шаг (следующий цикл логики) используем ключевое слово *yield*, или конструкцию *yield from*

Кроме main есть еще методы-генераторы: *until()*, *till()*, *pause()*. Также можно создавать и использовать свои. Выполнение методов-генераторов можно запустить параллельно основной логике с помощью *exec()*, и остановить в любое время.

Пример:

```
from pyplc.config import plc
from pyplc.sfc import SFC,POU

class UserProg(SFC):
   def __init__(self, id: str = None, parent: POU = None) -> None:
      super().__init__(id, parent)

   def main(self):
      self.log('красный')
      yield from self.pause(15000)
      self.log('желтый')
      yield from self.pause(3000)
      self.log('зеленый')
      yield from self.pause(30000)

user_prog = UserProg()

plc.run(instances=[user_prog],ctx=globals())
```

При запуске в консоли должно появиться что-то вроде

```
Loading PyPLC version v0.1.8-g2edad98:  simulation mode.
Initialized PYPLC with scan time=100 msec!
[1033] #user_prog   : красный
[16039] #user_prog   : желтый
[19058] #user_prog   : зеленый
[49168] #user_prog   : красный
```

метод `log()` выводит в скобках [] в начале строки время в мсек вызова, затем id программы (параметр id конструктора), затем после двоеточия идет пользовательское сообщение.

`class pyplc.sfc.SFC(`*id: str | None = None*, *parent:* POU *| None = None*`)`

`exec(`*act*`)`

Создать действие и добавить его в фоновое выполнение (однократное выполнение).

Параметры
act (*callable | generator*)

Результат
созданный генератор

Тип результата
generator

`main()`

Пользовательская логика. Необходимо написать свою реализацию этого метода, используя в нем ключевое слово *yield* там где должен быть следующий шаг.

`pause(`*T: int*, *step: str | None = None*, *n=[]*`)`

Пауза в программе на T мсек.

Пример использования:
yield from self.pause(1000,step="пауза 1 сек")

Параметры

- T (*int*) – мсек
- step (*str, optional*) – комментарий. Defaults to None.

till(*cond: callable*, *min: int | None = None*, *max: int | None = None*, *step: str | None = None*, *enter: callable | None = None*, *exit: callable | None = None*, *n=[]*)

Выполнять пока выполняется условие

Пример использования:
yield from self.till(lambda: self.clk, min = 1000, step = „ждем пока clk не станет False“)

Параметры

- cond (*callable*) – Логическое выражение, которое проверяется
- min (*int, optional*) – Минимальное время работы в мсек. Defaults to None.
- max (*int, optional*) – Максимальное время работы в мсек. Defaults to None.
- step (*str, optional*) – Пользовательский комментарий. Defaults to None.
- enter (*callable, optional*) – Что выполнить в начале. Defaults to None.
- exit (*callable, optional*) – Что выполнить в конце. Defaults to None.
- n (*list, optional*) – функции, которые пока выполнение происходит вызываются с True в качестве параметра, за затем с False. Defaults to [].

until(*cond: callable*, *min: int | None = None*, *max: int | None = None*, *step: str | None = None*, *enter: callable | None = None*, *exit: callable | None = None*, *n=[]*)

Выполнять пока не выполнится условие, противоположность *till()*

5.4 Библиотека функциональных блоков

5.4.1 Таймеры

Функциональные блоки для организации реле времени, генераторов импульсов заданной длинны.

TON([clk, q, et, pt, id, parent])	Задержка включения
TOF([clk, q, et, pt, id, parent])	Задержка при отключении
BLINK([enable, q, t_on, t_off, id, parent])	Включение/выключение на заданные интервалы.
TP([clk, t_on, t_off, q, id, parent])	Импульс указанной длины.

class pyplc.utils.misc.TON(*clk: bool = False*, *q: bool = False*, *et: int = 0*, *pt: int = 1000*, *id: str | None = None*, *parent:* POU *| None = None*)

Задержка включения

clk

вход, который с задержкой pt появится на выходе q

et

сколько времени прошло с момента установки clk

pt
: задержка в мсек

q
: выход блока

class pyplc.utils.misc.TOF(*clk: bool = False*, *q: bool = False*, *et: int = 0*, *pt: int = 1000*, *id: str | None = None*, *parent:* POU *| None = None*)

Задержка при отключении

clk
: вход, который с задержкой pt снимается с выхода q

et
: сколько времени прошло с момента установки clk

pt
: задержка в мсек

q
: выход блока

class pyplc.utils.misc.BLINK(*enable: bool = False*, *q: bool = False*, *t_on: int = 1000*, *t_off: int = 1000*, *id: str | None = None*, *parent:* POU *| None = None*)

Включение/выключение на заданные интервалы. На выход q генерируется меандр с указанными временными настройками

enable
: включить/выключить работу блока

q
: выход блока

t_off
: время выключения

t_on
: время включения

class pyplc.utils.misc.TP(*clk=False*, *t_on: int = 1000*, *t_off: int = 0*, *q: bool = False*, *id: str | None = None*, *parent:* POU *| None = None*)

Импульс указанной длины. По входу clk на выходе q генерируется импульс заданной длинны и паузой после.

clk
: вход блока

t_off
: минимальное время в выключенном состоянии

t_on
: время во включенном состоянии

5.4.2 Триггеры

Функциональные блоки для контроля за изменением фронта входа. Триггеры на один вызов устанавливают выход q при переходе из одного состояния в другое входа clk.

RTRIG([clk, q, id, parent])	Детектирование перехода clk из False в True.
FTRIG([clk, q, id, parent])	Детектирование перехода clk из True в False.
TRIG([clk, q, id, parent])	Детектирование перехода clk из одного состояния в другое.

5.4.3 Триггеры с фиксацией

Функциональные блоки триггеров-защелок RS/SR. При наличии или фронте входа clk выход q устанавливается и сбрасывается уже другим сигналом (reset). Триггеры имеют разный приоритет входов set и reset.

RS([reset, set, q, id, parent])	Триггер с приоритетным входом reset
SR([set, reset, q, id, parent])	Триггер с приоритетным входом set

class pyplc.utils.latch.SR(*set=False*, *reset=False*, *q=False*, *id: str | None = None*, *parent:* POU | *None = None*)

Триггер с приоритетным входом set

q
: выход, состояние триггера

reset
: вход сброса q

set
: вход активация триггера

unset()
: Произвести сброс выхода q

class pyplc.utils.latch.RS(*reset=False*, *set=False*, *q=False*, *id: str | None = None*, *parent:* POU | *None = None*)

Триггер с приоритетным входом reset

q
: выход блока

reset
: вход сброса q. Пока ==True q = False

set
: вход активация триггера. q устанавливается по переходу из False в True

unset()
: Произвести сброс выхода q

Глава 6

Примеры использования

Первоначально KRAX PLC-932 создавался под задачу создания логики управления процессом приготовления бетона, сухих смесей. В качестве примера использования библиотеки PYPLC для написания программы можно ознакомиться с библиотекой concrete на github, в которой написаны функциональные блоки для бетонных заводов.

С примером применения этой библиотеки можно ознакомиться в примере логики управления Бетонным заводом (реально действующий объект).

Глава 7

Указатели и таблицы

- genindex
- modindex
- search

Содержание модулей Python

p

Алфавитный указатель

Символы

A

B

C

E

F

I

M

N

P

Q

R

S

T

U

W