Руководство пользователя KRAX932 для инженеров-программистов

Введение

Программирование контроллера производится на языке micropython (python 3 для микроконтроллеров). Для облегчения контроллер поставляется с предустановленным модулем PYPLC, который позволяет

- Программу на python реализовывать в стиле STL, SFC, LD.
- Обеспечить взаимодействие контроллера с компьютером для создания визуализации
- Производить мониторинг состояния программы
- Осуществлять доступ к каналам ввода-вывода

Общее описание ПЛК и его функций.

ПЛК необходим для подключения сигналов от датчиков и исполнительных механизмов и их обработки.

ПЛК состоит из одного процессорного модуля (PLC-932) и нескольких модулей расширения. Модули отличаются количеством каналов, типом подключаемого сигнала. Например DO530- 8 канальный дискретный выход 24В, DI430 — 8 канальный дискретный вход 24В, AI455 — 4 канальный токовый вход 4-20 мА и др. Модули и процессор обмениваются данными по беспроводному интерфейсу, могут быть установленны на некотором удалении друг от друга.

Цель руководства и его предполагаемая аудитория.

Документ предназначен для инженеров и программистов, с минимальными навыками программирования на каком либо языке программирования МЭК или классическими.

Мы попробуем показать на примерах, как сделать первые шаги в написании логики управления встречающимися в автоматизации управления оборудованием.

Обзор системы

Описание архитектуры ПЛК. Как ранее упоминалось ПЛК имеет модульную архитектуру. Может состоять из минимум одного процессорного блока KRAX-PLC932. Процессорный блок не имеет клеммников для подключения сигналов от оборудования, и для того чтобы что то подключить нужно использовать подходящий модуль расширения. Модули обмениваются данными по беспроводному интерфейсу, поэтому для подключения к процессору необходимо произвести «сопряжение», чем то похожее на подключение наушников AirPods/EarsBuds к телефону.
Также можно использовать встроенные интерфейсы Ethernet/Bluetooth/WiFi для управления/взаимодействия с сторонними устройствами, но это за рамками этого документа.

Установка и настройка

Подготовка к программированию контроллера состоит из следующих этапов:

- Составляем перечень сигналов, которые необходимо подключить и обработать к ПЛК. Перечень должен содержать тип сигнала (дискретный вход/выход, аналоговый вход/выход)
- Подбираем необходимое количество модулей соответствующих типов, чтобы можно было подключить сигналы из перечня.
- Подключаем питание на все модули и проводим процедуру сопряжения.
- Пишем программу, отлаживаем
- Загружаем в контроллер.

Сопряжение модулей

1. Удерживайте кнопку WPS на каждом модуле, пока синий индикатор не начнёт мигать (порядок не важен, но лучше начать с контроллера). Это мы вошли в режим сопряжения.
2. Нажмите и отпустите (коротко) кнопку WPS на контроллере. Моргнёт красный индикатор ERR, а на модулях - индикатор приёма данных COMM. Частота мигания индикатора на модулях увеличится. Если необходимо добавить еще модуль, можно как в п.п.1 ввести в режим сопряжения и повторить п.п.2
3. Нажмите кнопку WPS на первом модуле, который вы хотите подключить к контроллеру. На контроллере моргнёт индикатор приёма данных COMM. Если на контроллере не моргнул COMM (или не заметили) можно повторять этот шаг сколько угодно раз.
4. Повторите шаг 3 для всех остальных модулей в порядке подключения. Если индикатор получения данных контроллера не моргнул, вы можете нажать кнопку WPS на модуле ещё раз.
5. После того, как вы нажали кнопку WPS на последнем модуле, нажмите и удерживайте WPS на контроллере, пока синий индикатор не начнёт мигать, и отпустите. Модули перестанут мигать, и сопряжение будет завершено.

Выйти из процедуры сопряжения без внесения изменений в настройки можно нажав и удерживая WPS более 5 секунд, пока синий индикатор не начнет мигать с удвоенной частотой.

Программирование ПЛК

Контроллер включён и сопряжен с модулями. Приступим к написанию нашей первой программы. HelloWorld так сказать.
Программа контроллера — это файл(ы) на языке python и может быть 2 дополнительных файла настроек (используются модулем PYPLC): krax.json & io.csv.
После включения и загрузки системы контроллера запускается main.py. Изначально в этом файле проверяется состояние DIP переключателя RUN, и если он включён - запускается сценарий task.py
Все программы управления цикличны и в общих чертах представляют из себя последовательность: получить состояние с входных сигналов (состояние подключенных датчиков) — обработать их и выдать если необходимо новые сигналы на выходные сигналы — и так по кругу.

Пример №1: HelloWorld

Напишем и разберём программу, которая мигает светодиодом RUN каждый цикл работы. Стиль написания: Structured Text-подобный.

task.py

```
#Моргаем светодиодом RUN
from pyplc.config import plc,board

def hello():
    board.run = not board.run

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

Первая строчка подключает из модуля pyplc (в него входит подмодуль kx) и импортирует из него 2 объекта: plc и board. Первый нужен всегда, второй (board) имеет атрибуты, которые отвечают за светодиоды(он нам как раз нужен).
Далее идет объявление функции с именем hello, в теле которой атрибуту run присваивается логическое отрицание его текущего значения. board.run отвечает за светодиод RUN, имеет тип bool (логический, может быть False-выключен или True — включён).
Последняя строка используя объект plc запускает цикл работы программы. Для этого вызывается метод run с 2 параметрами: instances и ctx.

- Instances имеет тип списка (встроенный тип python). Все элементы этого списка вызываются каждый цикл работы программы. Мы в списке только одну функцию указали (обратите внимание, что мы не результат выполнения функции, а саму функцию передали)
- ctx имеет тип словаря (dict), и мы ему присваиваем словарь глобальных переменных текущего контекста. Немного мудрено, но только на первый взгляд. Разберём позже, в нашем примере можно было опустить.

К следующей строке программы, даже если бы она была выполнение дойдёт только в случае прерывания цикла работы (в случае ошибки, исключения или по команде).

Пример №2: Включить исполнительный механизм по срабатыванию датчика

Такая задача встречается как вспомогательная, например в случае набора расходной емкости водой по нижнему уровню. В нашем случае нам необходимо подключить 1 дискретный вход (сигнал с датчика) и 1 дискретный выход (управление механизмом, например насосом). Для этого потребуется 8-ми канальный KRAX DI-430 и 8-ми канальный KRAX DO-530. С лёгким избытком.

Теперь пишем код.

task.py

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

В отличии от первого примера нам понадобится доступ каналам ввода/вывода, их подключаем во второй строке.
В третьей строке подключаем объекты, которые позволяют писать код, напоминающий Ladder Diagram.

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

Объявили наши переменные, значение которых синхронизируется с каналами модулей ввода вывода. Для того чтобы объявить переменную ввода вывода мы выбираем ей имя (SWITCH_ON_1 & POWER_ON_1), и присваиваем им результат выполнения метода at соответствующего типа. Например дискретный вход это IBool, дискретный выход QBool. В качестве параметра указываем строку, которая указывает по какому адресу в памяти ввода вывода она находится. Подробную информацию об адресации каналов ввода вывода см в главе [название_придумай].

```
# пример программы в стиле Ladder Diagram.
#                SWITCH_ON_1       POWER_ON_1
#           ├──────────┤├──────────────────(S)──────────┤
turn_on = LD.no(SWITCH_ON_1).set(POWER_ON_1).end()
```

Здесь создаем программу turn_on. Комментарии специально сохранены, чтобы показать наглядно сходство кода на python и Ladder Diagram. Находите их похожими?

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

Тут тоже почти без изменений, имя функции только изменилось. Обратите внимание, turn_on тоже использована как функция. Дело в том, что в python объекты могут иметь функцию вызова, то-есть можно написать turn_on()- это будет вызов нашей логики один раз. А просто turn_on — это наша программа.

Необходимо отметить, что переменные ввода-вывода (SWITCH_ON_1 & POWER_ON_1) они представляют механизм доступа к физическим сигналом, но это тоже объекты. И это может приводить к путанице. Например код

```
if SWITCH_ON_1: POWER_ON_1 = True
```

сработает не как ожидается. Первая часть (if SWITCH_ON_1:) работает вполне ожидаемо, тоесть проверит какое состояние дискретного сигнала SWITCH_ON_1 и если он True, то выполнит код после :, а именно POWER_ON_1 = True. Можно ожидать, что наш дискретный выход будет установлен в True, но в дествительности наш объект POWER_ON_1 будет заменен на новый объект, тип которого будет bool. Такая особенность python, ни один раз приведет в замешательство. Чтобы избежать этого, лучше к переменным ввода-вывода обращаться через атрибуты объекта hw, который находится в pyplc.config. Наш пример будет выглядеть так:

```
if SWITCH_ON_1: hw.POWER_ON_1 = True
```

Немного непривычно, но только впервое время. Еще возможны следующие способы изменения значения переменных вывода, все равнозначные:

```
POWER_ON_1(True)
POWER_ON_1.write(True)
hw.POWER_ON_1 = True
```

Смотрите подробную информацию по работе с переменными вводы-вовода в главе [название_придумай]

Пример №3. Дискретный вход, выход и аналоговый вход.

Задача: включить по дискретному входу №1, выключить по входу №2 или уровню аналогового сигнала. Потребуется еще модуль поключения токового сигнала 4-20 мА KRAX-AI455. Он 4-х канальный. Порядок сопряжения с модулями такой: 932 — 455 — 430 — 530.

task.py

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

Добавось подключение hw и IWord (аналоговый канал)

```
SWITCH_ON_1 = IBool.at( '%IX8.0' )
SWITCH_OFF_1= IBool.at( '%IX8.1' )
LEVEL_1 = IWord.at( '%IW0' )
POWER_ON_1 = QBool.at( '%QX9.0' )
```

Объявили наши переменные, значение которых синхронизируется с каналами модулей ввода вывода. LEVEL_1 для работы с аналоговым входом.

```
# пример программы в стиле Ladder Diagram.
#                SWITCH_ON_1       POWER_ON_1
#           ├──────┤├──────────────────(S)──────┤
turn_on = LD.no(SWITCH_ON_1).set(POWER_ON_1).end()
```

Здесь создаем программу turn_on.

```
# в Ladder Diagram аналога нет, можно описать как несколько паралельных
# цепей (тут 2), и результатом их выполнения будет True если они все False
#
# SWITCH_OFF_1                                                    POWER_ON_1
# ├──┤ ├──────────────┬──┤NOT├──────────────────────────────────────(R)──────┤
# LEVEL_1>32767      │
# ├──┤ ├──────────────┘
turn_off=LD.nor(LD.no(SWITCH_OFF_1),LD.no(lambda:LEVEL_1>32767)).rst(POWER_ON_1).end()
```

Первая особенность. В качестве условия LEVEL_1>32767 используется lambda: LEVEL_1>32767 Если написать LEVEL_1>32767, то результат будет вычислен сразу же, будет True если в момент вычисления LEVEL_1>32767 или False иначе. А нам надо каждый цикл работы перевычислять. Для этого нужно создать функцию

```
def check_level()->bool:
    return LEVEL_1>32767
```

и её поставить. Не очень удобно создавать функцию для любого выражения, ей ещё имя придумать надо. На счастье в python есть возможность создавать анонимные функции, используя ключевое слово lambda. Этим пользуемся.

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

Пример №4. Аналоговый вход, дискретный выход и функциональные блоки.

В модуле pyplc есть наиболее необходимые блоки, посмотрим как их можно использовать.

Задача: включить исполнительный механизм если уровень на аналоговом входе >75%, выключить если <25%.

task.py

```
from pyplc.config import plc,hw
from pyplc.utils.latch import RS
from pyplc.channel import QBool,IWord
```

Нам потребуется блок RS. Это такая программа, с 2-мя входами (set,rst) и одним выходом (q). Если set=True, то q становится True. Если rst=True, то q становится False.

```
LEVEL_1 = IWord.at( '%IW0' )
POWER_ON_1 = QBool.at( '%QX8.0' )
```

Объявили наши переменные, значение которых синхронизируется с каналами модулей ввода вывода.

```
logic = RS(reset = lambda: LEVEL_1<0x3FFF, set = lambda: LEVEL_1>0x7FFF, q = POWER_ON_1 )
```

Здесь создаем программу logic. Настраиваем, что поступает ей на входы и куда выход подключить. На вход и выход все программы могут принимать функции, каждый цикл работы они будет вызываться. Причем функции которые мы подключаем к входам должны работать без агрументов, а те что подключаются к выходам напротив должны работать как минимум с одним аргументом. Мы создаем 2 безымянные функции с помощью ключевого слова lambda. А выход мы подключаем к POWER_ON_1, который как упоминалось выше тоже может использоваться как функция (POWER_ON_1(True) как раз устанавливает новое значение True). Если бы мы на set подключали дискретный вход (как в примере №3) SWITCH_ON_1, то использовать lambda нет необходимости. Было бы просто set = SWITCH_ON_1.

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

Блоки можно самостоятельно писать и потом их повторно использовать.

Пример №5. Создание блока и его использование

Расмотрим как создать свой блок и как его испоользовать. Назначение блока: включение двигателя с авто-подхватом, перед включением дать звонок 2 сек. Включение прозводится дискретным выходом длительностью не менее 1 сек. Выключение другим дискретным выходом длительностью не менее 1 сек.

Следовательно, наш блок должен иметь 2 выхода (ON,OFF) и один вход (MANUAL) для активации. Код файла motor.py:

```
from pyplc.sfc import *
```

Подключаем все из pyplc.sfc. Нужно для того чтобы писать в стиле SFC.

```
class Motor(SFC):
    manual = POU.input( False )
    ison = POU.input(False,hidden=True)
    on = POU.output(False,hidden=True)
    off= POU.output(False,hidden=True)
    bell = POU.output(False,hidden=True)
```

Объявляем наш класс. Он наследует поведение и атрибуты класса SFC (который в свою очередь потомок класса POU). И перечисляем его входы-выходы. Аттрибуты, которые объявлены через POU.input/output/var могут быть подключены к функциям.

```
@POU.init
def __init__(self,manual: bool= False, on:bool = False,off:bool = False,bell: bool=False ) -> None:
    super().__init__( )
    self.on = on
    self.off= off
    self.bell=bell
    self.manual = manual
```

Конструктор нашего класса. Специально параметры указаны с аннотацией типов (в python это необязательно), чтобы в редакторе Auto-Completer показывал подсказку что должны

возвращать/принимать подключаемые на вход/выход функции. Важным является использование декоратора @POU.init перед конструктором, тогда если на вход подается функция, она будет вычислена и в наш конструктор будет передан уже ее результат.

```
@sfcaction
def main(self):
    for x in self.until(lambda: self.manual,step='Ждем manual'):
        yield x
    self.bell = True
    for x in self.pause(2000,step='Звонок'):
        yield x
    self.bell = False
    self.on = True
    for x in self.till(lambda: self.manual,max=1000,step='Импульс ON'):
        yield x
    self.on = False
    for x in self.till(lambda: self.manual,step='Работаем'):
        yield x
    self.off = True
    for x in self.pause(2000,step='Импульс OFF'):
        yield x
    self.off = False
```

Тело нашей логики. Также используя декоратор @sfcaction. Он необходим для реализации поведения SFC программы и синхронизации наших входов/выходов с результатом прикрепленных функций.

Программа должна каждый цикл опроса принимать решение и выдавать его. А у нас при получении сигнала manual надо включить звонок, подождать 2 сек и выключить его. В python есть возможность использовать многопоточность, но есть более предпочтительный метод. SFC-стиль позволяет обойтись обычной однопоточной программой. За счет использования ключевого слова yield.

```
for x in self.until(lambda: self.manual,step='Ждем manual'):
    yield x
```

Мы организуем бесконечный цикл, пока функция lambda: self.manual возвращает False. А в теле этого цикла yield x. На этом месте python синхронизирует выходы нашего блока и продолжает выполнение других блоков (если бы они были). Затем производит опрос модулей расширения и возвращается обратно на следующую за yield строку. В участке выше следующая строка — возврат в начало цикла. Снова проверка и так пока на входе manual не появится True. Потом переходит к следующему участку

```
self.bell = True
for x in self.pause(2000,step='Звонок'):
    yield x
self.bell = False
```

В первой строке на выход bell подаем True, и потом снова конструкция с циклом на 2000 мсек. По окончанию которой на bell выдаем False. Обратите внимание: если self.bell = True поместить в тело цикла, то каждый цикл работы программы bell будет повторно устанавливаться. А так если к выходу bell подключен физический сигнал он изменится всего 2 раза (в начале и конце).

```
self.on = True
for x in self.till(lambda: self.manual,max=1000,step='Импульс ON'):
```

```
    yield x
self.on = False
```

В этом участке кода как и со звонком, только цикл проверяет, не передумал ли пользователь и время выполнение цикла ограничено временем 1000 мсек. Когда функция main дойдет до конца, она начнет свое выполнение сначала.
Пример использования нашего блока:

```
from pyplc.config import plc
from pyplc.channel import IBool,QBool
from motor import Motor

SWITCH_ON_1 = IBool.at( '%IX0.0' )
POWER_ON_1 = QBool.at( '%QX1.0' )
POWER_OFF_1 = QBool.at( '%QX1.1' )
BELL_ON_1 = QBool.at( '%QX1.2' )

motor = Motor(manual=SWITCH_ON_1,on=POWER_ON_1,off=POWER_OFF_1, bell = BELL_ON_1)
plc.run(instances=[motor],ctx=globals())
```

Просто, как два байта переслать.

Тестирование и отладка

Для тестирования и отладки используется следующая методика: программа пишется, отлаживается на компьютере (без контроллера), так как синтаксис Python и MicroPython одинаковый, модуль pyplc написан на python и работает на любом ПК. В качестве среды разработки используется VSCode с дополнением для обмена файлами с контроллером KRAX.
В VSCode доступны все привычные методы отладки (контрольные точки, просмотр значений любой переменной, пошаговое выполнение и т. п.). Модуль PyPLC позволяет прозрачно подключиться к контроллеру (если он есть), и тогда сигналы с датчиков и управляющие сигналы синхронизируются.

Советы по поиску и устранению проблем.

1. Каждый цикл работы программы моргайте индикатором RUN. Это позволит с первого взгляда определить работает программа сейчас или закончила своё выполнение.
2. Пользуйтесь выводом на консоль (функция print).
3. Если надо проверить состояние переменных во время выполнения программы в контроллере можно воспользоваться telnet подключением (доступно на порту 2455). После установки соединения можно написать выражение на python и оно будет выполнено или вычислено. Например если написать board.run вам в ответ будет показано текущее состояние светодиода RUN.

Приложения

[такие как ссылки на онлайн-ресурсы, библиотеки кода и т.д.]

Заключение

Контактная информация для поддержки.