УТВЕРЖДЕНО

РОФ.МГТУ.000001 12

СТОРОЖЕВОЙ ТАЙМЕР

Текст программы

РОФ.МГТУ.000001-01 12

Инв. № подл.	Подпись и дата	Взам. инв. №	Инв. № дубл.	Подпись и дата

АННОТАЦИЯ

В данном программном документе приведен текст программы для терминала самообслуживания. Текст программы реализован в виде символической записи на исходном языке. Исходным языком данной разработки являются Python и Си. Среды разработки: PyCharm 2021.3 и MPLAB X IDE 3.14. Интерпретатор: Python 3.8.0. Компилятор: CCS C Compiler 5.114.

Основной функцией программы USB WatchDog Agent является конфигурирование и настройка сторожевого таймера для перезапуска системы и приложений.

Основной функцией программы WatchDog Timer является принятие сигналов сброса и конфигурации, обновление таймера и подача сигнала RESET в случае истечения времени таймера.

Оформление программного документа «Текст программы» произведено по требованиям ЕСПД (ГОСТ 19.101-77 [1], ГОСТ 19.103-77 [2], ГОСТ 19.104-78* [3], ГОСТ 19.105-78* [4], ГОСТ 19.106-78* [5], ГОСТ 19.401-78 [6], ГОСТ 19.604-78* [7]).

---

[1] ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов
[2] ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов
[3] ГОСТ 19.104-78* ЕСПД. Основные надписи
[4] ГОСТ 19.105-78* ЕСПД. Общие требования к программным документам
[5] ГОСТ 19.106-78* ЕСПД. Общие требования к программным документам, выполненным печатным способом
[6] ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению
[7] ГОСТ 19.604-78* ЕСПД. Правила внесения изменений в программные документы, выполненные печатным способом

1. ТЕКСТ ПРОГРАММЫ USB WHATCHDOG AGENT НА ИСХОДНОМ ЯЗЫКЕ

main.py

```
1. from concurrent.futures import ThreadPoolExecutor
2.
3. from agent.watchdogapp import WatchDogApp
4.
5. if __name__ == '__main__':
6.     app = WatchDogApp()
7.     with ThreadPoolExecutor() as pool:
8.         pool.submit(app.listening)
9.         pool.submit(app.check_targets)
10.          pool.submit(app.run())
```

watchdogapp.py

```
1. import tkinter as tk
2. import psutil
3.
4. from time import sleep
5.
6. from agent.utlis import run_app
7. from agent.frames import (
8.     ComChoosingFrame,
9.     TimerConfigFrame,
10.     TargetedAppsFrame,
11.     connected_port
12. )
13.
14. BACKGROUND = '#D3D3D3'
15.
16.
17. class WatchDogApp:
18.     TITLE = 'USB WatchDog Agent v.1.0.0'
19.
20.     def __init__(self):
21.         self.root = tk.Tk()
22.         self.is_running = True
23.         root = self.root
24.         root.title(self.TITLE)
25.         root.resizable(width=False, height=False)
26.         root.protocol('WM_DELETE_WINDOW', self.on_exit)
27.
28.         self.com_choosing_frame = ComChoosingFrame(
29.             root, borderwidth=5, background=BACKGROUND, border=1
30.         )
31.
```

```
32.         self.timer_config_frame = TimerConfigFrame(
33.             root, borderwidth=5, background=BACKGROUND, border=1
34.         )
35.
36.         self.targeted_apps_frame = TargetedAppsFrame(
37.             root, borderwidth=5, background=BACKGROUND, border=1
38.         )
39.
40.         self.com_choosing_frame.grid(row=0, sticky='WE')
41.         self.timer_config_frame.grid(row=1, sticky='WE')
42.         self.targeted_apps_frame.grid(row=2, sticky='WE')
43.
44.     def run(self):
45.         self.is_running = True
46.         self.root.mainloop()
47.
48.     def on_exit(self):
49.         self.is_running = False
50.         self.root.destroy()
51.
52.     def check_targets(self):
53.         while self.is_running:
54.             running_proc_names = {
55.                 process.name()
56.                 for process in psutil.process_iter()
57.             }
58.             target_processes = self.targeted_apps_frame.target_processes
59.             for name_process, exe_cmdline in target_processes.items():
60.                 if name_process not in running_proc_names:
61.                     run_app(exe_cmdline[0])
62.
63.             sleep(3)
64.
65.     def listening(self):
66.
67.         while self.is_running:
68.             if connected_port is None:
69.                 continue
70.
71.             if not connected_port.is_open:
72.                 connected_port.open()
73.
74.             connected_port.write()
75.
76.             sleep(1)
77.             while connected_port.inWaiting() > 0:
78.                 data = connected_port.readline()
```

```
79.                 print(data)
```

serial_com.py

```
1. from serial.tools import list_ports
2.
3.
4. def get_ports() -> list:
5.     """Gets list of available COMs."""
6.     ports = list_ports.comports()
7.     return [
8.         f'{port}: {desc}'
9.         for port, desc, _ in sorted(ports)
10.     ]
```

frames.py

```
1. import tkinter as tk
2.
3. from typing import Optional
4. from serial import Serial
5. from tkinter.ttk import Combobox
6. from tkinter.messagebox import showerror
7.
8. from agent.utlis import get_process_dict, restart_app
9. from agent.serial_com import get_ports
10.
11. BACKGROUND = '#D3D3D3'
12.
13. connected_port: Optional[Serial] = None
14.
15.
16. class ComChoosingFrame(tk.Frame):
17.     NOT_CHOSEN = 'Не выбрано'
18.     CONNECTED = 'подключено'
19.     NOT_CONNECTED = f'не {CONNECTED}'
20.
21.     def __init__(self, *args, **kwargs):
22.         super().__init__(*args, **kwargs)
23.
24.         tk.Label(
25.             self, text='Подключение к устройству', font=20,
26.             background=BACKGROUND
27.         ).grid(row=0, column=0, padx=1, pady=1, sticky=tk.W)
28.
29.         tk.Label(
30.             self, text='Serial:', background=BACKGROUND
31.         ).grid(row=1, column=0, sticky=tk.W, pady=1)
```

```
32.
33.         self.available_coms = Combobox(
34.             self, values=self.combobox_values, width=50,
    state='readonly',
35.         )
36.         self.available_coms.grid(
37.             row=2, column=0, sticky=tk.W, padx=2, pady=1
38.         )
39.         self.available_coms.current(0)
40.         self.available_coms.bind('<<ComboboxSelected>>',
    self.com_selected)
41.
42.         tk.Button(
43.             self, text='Сканировать', command=self.update_com_ports,
44.         ).grid(row=2, column=1, padx=2, pady=1)
45.
46.         self.status = tk.StringVar(value=f'Статус:
    {self.NOT_CONNECTED}')
47.         tk.Label(
48.             self, textvariable=self.status, background=BACKGROUND
49.         ).grid(row=3, column=0, padx=2, pady=1, sticky=tk.E)
50.
51.         self.connect_btn = tk.Button(
52.             self, text='Подключиться', command=self.connect,
    state='disabled',
53.         )
54.         self.connect_btn.grid(row=3, column=1, padx=2, pady=3)
55.
56.         self.disconnect_btn = tk.Button(
57.             self, text='Отключиться', command=self.disconnect,
    state='disabled',
58.         )
59.         self.disconnect_btn.grid(row=4, column=1, padx=2, pady=3)
60.
61.     def update_com_ports(self) -> None:
62.         """Updates list of available COMs in combobox."""
63.         self.available_coms.configure(values=self.combobox_values)
64.
65.     @property
66.     def combobox_values(self):
67.         return [self.NOT_CHOSEN] + get_ports()
68.
69.     def com_selected(self, event):
70.         if self.available_coms.get() != self.NOT_CHOSEN:
71.             self.connect_btn.config(state='normal')
72.             return
73.
74.         self.connect_btn.config(state='disabled')
```

```
75.
76.     def connect(self):
77.         chosen = self.available_coms.get()
78.         if chosen == self.NOT_CHOSEN:
79.             showerror(
80.                 title='Упс!', message='Вы не выбрали подходящий COM.'
81.             )
82.             return
83.
84.         global connected_port
85.         try:
86.             name = chosen.split(':')[0]
87.             connected_port = Serial(name, 9600)
88.         except Exception as err:
89.             connected_port = None
90.             self.status.set(f'Статус: {self.NOT_CONNECTED}')
91.             showerror('Критическая ошибка!', str(err))
92.             return
93.
94.         self.status.set(f'Статус: {self.CONNECTED}')
95.         self.disconnect_btn.config(state='normal')
96.
97.         if connected_port is None:
98.             print(f'{connected_port} is None')
99.
100.        if not connected_port.is_open:
101.            connected_port.open()
102.
103.        connected_port.write()
104.
105.    def disconnect(self):
106.        global connected_port
107.
108.        try:
109.            connected_port.close()
110.            connected_port = None
111.            self.status.set(f'Статус: {self.NOT_CONNECTED}')
112.            self.disconnect_btn.config(state='disabled')
113.        except Exception as err:
114.            self.status.set(f'Статус: {self.NOT_CONNECTED}')
115.            showerror('Критическая ошибка!', str(err))
116.            return
117.
118.
119. class TimerConfigFrame(tk.Frame):
120.    def __init__(self, *args, **kwargs):
121.        super().__init__(*args, **kwargs)
122.        tk.Label(
```

```
            self, text='Конфигурация таймера', font=20,
    background=BACKGROUND
        ).grid(columnspan=3, row=0, column=0, padx=1, pady=1,
    sticky=tk.W)

        tk.Label(
            self, text='Время сброса:', background=BACKGROUND
        ).grid(row=1, column=0, sticky=tk.W, pady=1)

        Combobox(
            self, values=tuple(str(i + 1) for i in range(5)),
    state='readonly',
        ).grid(row=1, column=1, padx=2, pady=1)
        tk.Label(
            self, text='мс', background=BACKGROUND
        ).grid(row=1, column=2, pady=1)


class TargetedAppsFrame(tk.Frame):
    target_processes = {}

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        tk.Label(
            self, text='Отслеживание процессов', font=20,
            background=BACKGROUND
        ).grid(columnspan=3, row=0, column=0, padx=1, pady=2,
    sticky=tk.W)

        tk.Label(
            self, text='Введите имя процесса:', background=BACKGROUND
        ).grid(row=1, column=0, sticky=tk.W, padx=1, pady=2)

        self.entry = tk.Entry(self)
        self.entry.grid(row=1, column=1, pady=2)
        self.entry.bind('<KeyRelease>', self.search_entry)

        self.add_target_btn = tk.Button(
            self, text='Отслеживать', command=self.add_target,
    state='disabled'
        )
        self.add_target_btn.grid(row=1, column=2, sticky=tk.E, pady=10)

        ############################

        tk.Label(
            self, text='Запущенные процессы:', background=BACKGROUND
        ).grid(row=2, column=0, sticky='SW')
```

```
166.
167.         tk.Button(
168.             self, text='Обновить', command=self.scan_processes,
169.         ).grid(row=2, column=1, padx=5, pady=1, sticky=tk.E)
170.
171.         self.listbox_processes = tk.Listbox(self, width=50,
    relief=tk.RAISED)
172.         self.listbox_processes.grid(columnspan=2, row=3, column=0,
173.                                     padx=5, pady=3)
174.         self.listbox_processes.bind('<<ListboxSelect>>', self.fill_out)
175.
176.         self.processes_dict = get_process_dict()
177.         self.update_listbox(self.processes_list)
178.
179.         ############################
180.
181.         tk.Label(
182.             self, text='Отслеживаемые процессы:', background=BACKGROUND
183.         ).grid(row=4, column=0, sticky='SW')
184.
185.         self.listbox_targets = tk.Listbox(
186.             self, width=50, relief=tk.RAISED, selectmode=tk.EXTENDED
187.         )
188.         self.listbox_targets.grid(
189.             columnspan=2, row=5, column=0, padx=5, pady=3
190.         )
191.         self.listbox_targets.bind(
192.             '<<ListboxSelect>>', self.raise_buttons
193.         )
194.
195.         self.restart_target_btn = tk.Button(
196.             self, text='Рестарт', command=self.target_restart,
    state='disabled'
197.         )
198.         self.restart_target_btn.grid(
199.             row=5, column=2, padx=5, pady=1,
200.         )
201.
202.         self.remove_target_btn = tk.Button(
203.             self, text='Удалить', command=self.remove_target,
    state='disabled'
204.         )
205.         self.remove_target_btn.grid(
206.             row=6, column=1, padx=5, pady=1, sticky=tk.E
207.         )
208.
209.     @property
210.     def processes_list(self):
```

```
211.         return list(self.processes_dict.keys())
212.
213.     def scan_processes(self):
214.         self.processes_dict = get_process_dict()
215.         self.update_listbox(self.processes_list)
216.         self.entry.delete(0, tk.END)
217.
218.     def fill_out(self, event):
219.         self.entry.delete(0, tk.END)
220.         self.entry.insert(0, self.listbox_processes.get(tk.ANCHOR))
221.         self.add_target_btn.config(state='normal')
222.
223.     def update_listbox(self, process_list: list):
224.         self.listbox_processes.delete(0, tk.END)
225.         for process_name in process_list:
226.             self.listbox_processes.insert(tk.END, process_name)
227.
228.     def search_entry(self, event) -> None:
229.         self.add_target_btn.config(state='disabled')
230.         typed = self.entry.get()
231.         if not typed:
232.             self.update_listbox(self.processes_list)
233.             return
234.
235.         self.update_listbox([
236.             process_name
237.             for process_name in self.processes_list
238.             if typed.lower() in process_name.lower()
239.         ])
240.
241.     def add_target(self):
242.         process_name = self.entry.get()
243.         exe_path, cmdline = self.processes_dict[process_name]
244.         if process_name in self.target_processes:
245.             showerror(
246.                 title='Внимание',
247.                 message='Данный процесс уже отслеживается!'
248.             )
249.         else:
250.             self.listbox_targets.insert(tk.END, process_name)
251.             self.target_processes[process_name] = (exe_path, cmdline)
252.
253.         self.entry.delete(0, tk.END)
254.         self.search_entry(event=None)
255.
256.     def remove_target(self):
257.         process_name = self.listbox_targets.get(tk.ANCHOR)
258.         self.target_processes.pop(process_name)
```

```
259.         self.listbox_targets.delete(tk.ANCHOR)
260.         self.disable_buttons()
261.
262.     def raise_buttons(self, event):
263.         self.remove_target_btn.config(state='normal')
264.         self.restart_target_btn.config(state='normal')
265.
266.     def disable_buttons(self):
267.         self.remove_target_btn.config(state='disabled')
268.         self.restart_target_btn.config(state='disabled')
269.
270.     def target_restart(self):
271.         process_name = self.listbox_targets.get(tk.ANCHOR)
272.         exe_path, _ = self.processes_dict[process_name]
273.         kill_code, process_open = restart_app(process_name, exe_path)
274.         if kill_code != 0:
275.             showerror(
276.                 title='Увы и ах!',
277.                 message='Не удалось "убить" процесс :('
278.             )
279.             return
280.         if process_open is not None:
281.             showerror(
282.                 title='Увы и ах!',
283.                 message='Процесс убит, но не перезапущен :('
284.             )
285.             return
286.
287.         self.remove_target()
```

utils.py

```
1. import subprocess
2. import psutil
3.
4. from typing import Optional, Dict, Tuple, List
5.
6.
7. def restart_app(exe_path: str, cmdline: str) -> Tuple[int, int]:
8.     """
9.     Forced kills the process, with children and restarts it again.
10.      :param exe_path: path to process to kill
11.      :param cmdline: cmdline to restart the process
12.      :return: tuple of status codes of killing and run commands.
13.      """
14.      kill_process = subprocess.call(['TASKKILL', '/F', '/T', '/IM',
    exe_path])
15.      restart_process = run_app(cmdline)
```

```
    return kill_process, restart_process

def run_app(cmdline: str) -> int:
    """
    Runs app by command line.
    :param cmdline: cmdline to run the process.
    :return: status code.
    """
    return subprocess.Popen(cmdline).returncode

def get_process_dict() -> Dict[str, Tuple[str, List[str]]]:
    """
    Collects the dict of running processes.
    process_name -> Tuple[exe_path, cmdline]
    :return: resulted dict of processes
    """
    processes: Dict[str, Tuple[str, List[str]]] = {}
    for process in psutil.process_iter():
        process_params = __get_params(process)
        if process_params is not None:
            processes[process_params[0]] = (  # process_name
                process_params[1],  # exe_path
                process_params[2],  # cmdline
            )

    return processes

def __get_params(
        process: psutil.Process
) -> Optional[Tuple[str, str, List[str]]]:
    try:
        process_name = process.name()
        exe_path = process.exe()
        cmdline = process.cmdline()
        is_none_or_empty = any([
            attr is None or not attr
            for attr in (process_name, exe_path, cmdline)
        ])
        if is_none_or_empty or
exe_path.startswith('C:\\Windows\\System'):
            return None

        return process_name, exe_path, cmdline
    except psutil.AccessDenied:
        return None
```

2. ТЕКСТ ПРОГРАММЫ WHATCHDOG TIMER НА ИСХОДНОМ ЯЗЫКЕ

main.h

```
#include <18F46K22.h>
#device ADC=10

#FUSES NOWDT //No Watch Dog Timer
#use delay(crystal=8MHz,restart_wdt)
#use FIXED_IO(D_outputs=PIN_D0)

#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, bits=8, parity=N)
```

main.c

```
#include <main.h>
#include <ctype.h>

const char HEARTBEAT = 'h';
int timeout_error = 0;

char recieved = 0;
char second = 0;
unsigned int wdt_counter = 0;
unsigned int critical_time = 5; // 3 seconds by default

char timed_getc() {
    long timeout;
    timeout_error=FALSE;
    timeout = 0;
    while (!kbhit() && (++timeout < 50000)) // 1/2 second
      delay_us(10);
   if (kbhit()) {
     return (getc());
   } else {
     timeout_error=TRUE;
     return (0);
 }
}


unsigned int try_cast_integer(char first, char second) {
    if (isdigit((unsigned char)first) == 0) return -1;
    if (isdigit((unsigned char)second) == 0) return -1;

    return (first - '0') * 10 + (second - '0');
}

// Handles heartbeats and configuration from wdt_agent.
void UART_handler() {
```

```
      unsigned int time = 0;
      recieved = timed_getc();
      if (timeout_error == TRUE) {
       return;
      }
      putc(recieved);
      if (recieved != HEARTBEAT) {
         second = getc();
         putc(' ');
         putc(recieved);
         putc(second);
         putc(' ');

         time = try_cast_integer(recieved, second);
         if (time == -1) {
           puts("Îøèáêà!");
           return;
         }
         // Configure critical time from agent.
         critical_time = time;
         return;
      }
      // If heartbeat has been recieved, update wdt.
      wdt_counter = 0;
}

// Activates relay and restarts the system.
void reset() {
    output_high(PIN_D0);
    delay_ms(600);
    output_low(PIN_D0);
}

void main() {
   output_low(PIN_D0);
   // Initial heartbeat for starting process.
   for (getc(); TRUE; ++wdt_counter) {
     if (input(PIN_C7)) {
       UART_handler();
     }
     delay_ms(750);
     if (wdt_counter > critical_time) {
       reset();
       wdt_counter = 0;
       delay_ms(60000); // wait a minute, system restarts...
     }
   }
}
```