0.0.1. Obsługa modułu SIM800

Do obsługi modułu przygotowano bibliotekę przeprowadzającą całą komunikację. Ponieważ dane z modułu mogą napływać w każdym momencie (dotyczy to głównie komunikacji poprzez TCP), biblioteka używa własnego, działającego w tle, wątku (według terminologii *FreeRTOS* *zadania* (ang. *task*)).

0.0.1.1. Terminologia stosowana w bibliotece

Terminologia nazw funkcji inicjalizujących komendy AT (wyjaśnienie użycia funkcji w punkcie 0.0.1.3) bazuje na terminologii samych komend AT. Istnieją cztery rodzaje komend AT:

1. *Test* - zwraca listę parametrów i argumenty obsługiwane przez poszcególne komendy *Write*. Składnia komendy wygląda następująco: `AT+<x>=?`. `<x>` reprezentuje nazwę komendę (np. `CREG` - „*Network Registration*"). Ten rodzaj komend nie jest używany w bibliotece.
2. *Read* - zwraca aktualnie używany parametr lub parametry. Składnia: `AT+<x>?`.
3. *Write* - umożliwia użytkownikowi ustawienie paramteru/parametrów. Składnia: `AT+<x>=<...>`. `<...>` reprezentują argumenty wprowadzane przez użytkownika.
4. *Execute* - odczytuje parametry zależne od wewnętrznych procesów terminala, takie jak moc sygnału, numer CCID itp. Składnia: `AT+<x>`

Każda funkcja w bibliotece używa przedrostka: `SIM_`, dzięki czemu można uniknąć powtarzania nazw z funkcjami innych bibliotek. Następnie, w przypadku funkcji inicjalizujących komendy, określony jest rodzaj komendy:

- `read`,
- `write`,
- `exec`.

Ostatecznie występuje nazwa komendy `x`; Każda komenda przyjmuje wskaźnik do struktury `SM_cmd`, przechowującej nazwę komendy, wskaźniki do funkcji interpretujących komendy - funkcji *callback* (wyjaśnienie w punkcie 0.0.1.3) oraz obiektu `SIM_resp`, przechowującego wskaźniki do poszczególnych części otrzymanej odpowiedzi.

Zatem nazwy funkcji dla:

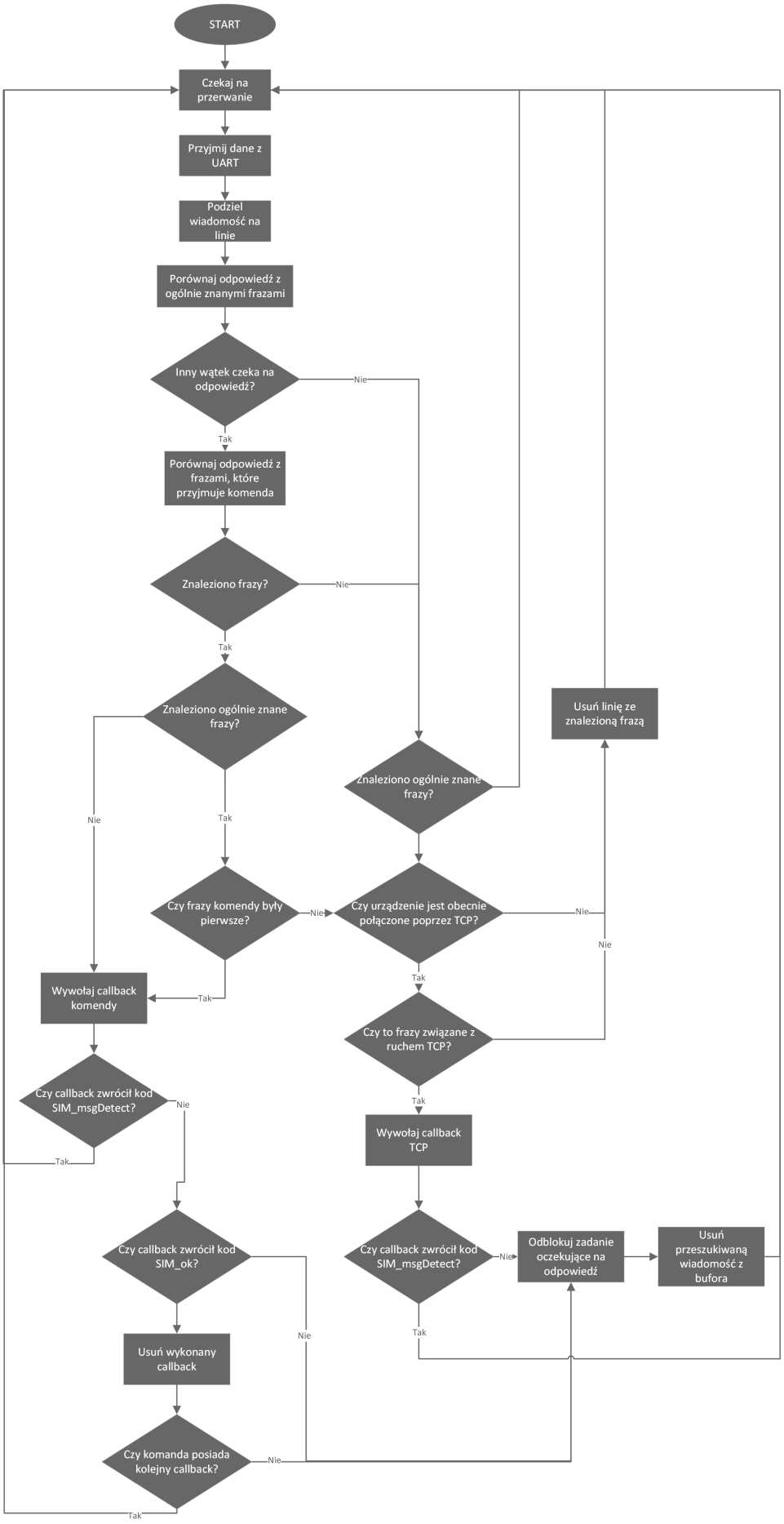
- *read*: `SIM_read<x>(SIM_cmd *)`,
- *write*: `SIM_write<x>(SIM_cmd *, <...>)`,
- *execute*: `SIM_exec<x>(SIM_cmd *)`.

0.0.1.2. Wątek główny biblioteki

Aby korzystać z biblioteki wymagane jest:

1. Stworzenie struktury `SIM_intf`, zawierającej informacje wymagane do sterowania modułem. Może zostać zainicjalizowany poprzez użycie funkcji `LL_SIM_def(SIM_intf *)`, która ustawia prędkość interfejsu UART na 115200 bps oraz numery pinów na zgodne z tymi wymienionym w punkcie **??**.
2. Inicjalizacja interfejsów metodą `LL_SIM_init(const SIM_intf *)`.
3. Stworzenie wątku `SIM800 deamon`, w którym wywoływana będzie cyklicznie funkcja `LL_SIM_receiveRaw(SIM_intf *)`, a następnie `SIM_exec(SIM_intf *)`. Funkcja `LL_SIM_receiveRaw(SIM_intf *)` oczekuje na przerwanie otrzymania danych od sterownika UART ESP32 oraz pobiera je. Następnie `SIM_exec(SIM_intf *)` zajmuje się interpretacją danych.

Rysunek 0.1 przedstawia logikę działania wątku z wykorzystaniem funkcji `LL_SIM_receiveRaw(SIM_intf *)` oraz `SIM_exec(SIM_intf *)`.



Rys. 0.1: Logika wątku `SIM800 deamon`

Interpretacja odpowiedzi na komendy AT polega na znajdowaniu określonych fraz w wiadomościach, które występują zawsze na początku linii. Koniec linii jest sygnalizowany poprzez znak końca linii („\n”).

Aby zmniejszyć nakład obliczeniowy wymagany do ich znalezienia, każda wiadomość jest dzielona na linie. Zatem dla wiadomości składającej się z N znaków, wymagane jest wykonanie N porównań w celu znalezienia wszystkich początków linii. Następnie każdy początek linii jest przeszukiwany pod kątem wystąpienia danej frazy. Zatem jeżeli w danej linii fraza nie występuje wystarczy wykonać zaledwie jedno porównanie pierwszego znaku frazy i początku pierwszego znaku nowej linii.

Podejście ma również zaletę, że jeżeli fraza wystąpi w wiadomości, ale nie na początku linii, wówczas nie będzie brana pod uwagę.

Poprzez „ogólnie znane frazy” biblioteka rozumie wiadomości, które mogą pojawić się w każdym momencie użycia modułu. Są one określone w tablicy `SIM_reservedResps` (listing 1), zakończonej znakiem `NULL` w celu zasygnalizowania końca tabeli. Frazy zostały wyznaczone eksperymentalnie podczas testowania modułu.

Listing 1: Ogólnie znane frazy

```
const SIM_err_pair SIM_reservedResps[] = {
                  {.name = "+RECEIVE", .err = SIM_receive},
                  {.name = ".␣CLOSED", .err = SIM_closed},
                  {.name = "SMS␣READY", .err = SIM_smsReady},
                  {.name = "Call␣READY", .err = SIM_callReady},
                  {.name = "RING", .err = SIM_ring},
                  {.name = NULL, .err = SIM_noErrCode}};
```

Znak „.” oznacza jeden dowolny symbol (zatem jeśli np. moduł wyśle następujący ciąg: „`0,`
↪ `CLOSED`”, to zostanie on zinterpretowany jako: „`., CLOSED`”).

Poprzez „frazy komendy” biblioteka rozumie określone przy definiowaniu komendy frazy. Jeśli nie zostały one określone, wówczas zakładane są tylko dwie:

- „`OK`”, oznaczająca powodzenie,
- „`ERROR`”, oznaczająca niepowodzenie.

Po wykonaniu zdefiniowanej przez komendę funkcji callback, deamon musi zdecydować jaką akcję podjąć na podstawie wartości zwracanej przez callback. Istnieją trzy scenariusze:

1. Zwrócona wartość to `SIM_ok`, zatem callback powiódł się. Jeżeli zdefiniowany jest następny callback, to on będzie użyty po otrzymaniu następnej porcji danych. Jeśli nie, wówczas można odblokować oczekujący wątek i przekazać wynik operacji.
2. Zwrócona wartość to `SIM_msgDetect` (ang. *Message Detected*). Frazy zostały znalezione, ale nie można dokończyć analizy, ponieważ wiadomość nie dotarła cała. Deamon musi poczekać na następną porcję danych i wtedy uruchomić ten sam callback.
3. Zwrócona została inna wartość. Jest to zawsze interpretowane jako niepowodzenie. Należy odblokować oczekujący wątek i przekazać wynik operacji.

0.0.1.3. Użycie biblioteki

Aby wysłać do modułu komendę potrzebne jest zdefiniowanie funkcji inicjalizującej strukturę komendy `SIM_cmd` oraz funkcję *callback*, sprawdzającą poprawność odpowiedzi odesłanej przez moduł. W bibliotece zdefiniowano tylko komendy, które potrzebne były w ramach projektu.

Przykładowo, listing 2 przedstawia funkcję inicjalizującą komendę `AT+CREG?`, odpowiadającą za sprawdzenie czy moduł zarejestrował się w sieci komórkowej.

Listing 2: Funkcja inicjalizująca komendę `AT+CREG?`

```
SIM_cmd *SIM_readCREG(SIM_cmd *cmd)
{
  /* Przygotuj komendę wysyłaną do modułu */
  SIM_param params[1];
  *params[0].name = '\0';
  SIM_setAT(cmd->at, "CREG?", params);

  /* Ustaw funkcję callback */
  cmd->handlers[0] = &SIM_readCREG_handler;
  cmd->handlers_num = 1;

  /* Wyczyść strukturę przechowującą odpowiedź od modułu */
  SIM_respNULL(&cmd->resp, cmd->at);

  /* Ustaw maksymalny czas oczekiwania na odpowiedź */
  cmd->timeout = SIM_READCREG_TIMEOUT;

  return cmd;
}
```

Funkcja przyjmuje wskaźnik do struktury `SIM_cmd`, zawierającej najważniejsze informacje o komendzie. Struktura jest następnie modyfikowana, wykonywane jest kolejno:

1. Ustawienie nazwy komendy przy pomocy funkcji `SIM_setAT`. Funkcja przyjmuje wskaźnik do tablicy znaków, gdzie zostanie umieszczona komenda (`cmd->at`), nazwę komendy (w powyższym przypadku: `"CREG?"`) oraz jej parametry w postaci tablicy struktur `SIM_param` (`params`), zawierających ciągi znaków (*cstring*). Jeżeli komenda używa N parametrów, wówczas potrzebna jest tablica wielkości $N + 1$, ponieważ ostatni parametr stanowi ciąg znaków o zerowej długości (tablica zawierająca jeden zank „`\0`”) i oznacza koniec tablicy. Ponieważ komenda `AT+CREG?` nie przyjmuje żadnych parametrów, tablica przetrzymuje jeden ciąg o zerowej długości.
2. Ustawienie funkcji *callback*, interpretującej wynik zapytania (w powyższym przypadku jest to funkcja `SIM_readCREG_handler`) oraz ilości używanych funkcji.
3. Wyczyszczenie struktury `SIM_resp` przechowywującej informacje o wyniku zapytania (za pośrednictwem funkcji `SIM_respNULL`).
4. Określenie maksymalnej długości oczekiwania na wynik zapytania, w *ms*.

Z dokumentacji można odczytać, że przykładowa, poprawna odpowiedź na komendę `AT+CREG?` może wyglądać następująco:

```
+CREG:1,1

OK
```

Dostarczana w trakcie definiowania komendy, funkcja callback ma za zadanie:

- sprawdzić poprawność wykonania komendy (sprawdzić czy moduł odesłał `"OK"`czy `"ERROR"`),
- ustawić wskaźniki na zwrócone parametry (w przypadku powyższej odpowiedzi parametry to: 1 oraz 1),
- wyznaczyć koniec odebranej odpowiedzi,
- wykonać inne operacje, jeżeli są one wymagane.

Wartości zwracane przez callback opiso w punkcie 0.0.1.2.

Listing 3 przedstawia callback, przetwarzający odpowiedź modułu na zapytanie `AT+CREG?`.

Listing 3: Funkcja callback komendy `AT+CREG?`

```
static SIM_error SIM_readCREG_handler(SIM_line_pair *lines, SIM_
   ↪ line_pair *lines_beg, SIM_resp *resp, void *sim)
{
  /* Sprawdź kod będu ("OK" czy "ERROR"?),
   * wyznacz koniec wiadomości
   */
  SIM_errMsgEnd_pair err = SIM_retrieveErr(lines);
  resp->err = err.err;
  resp->msg_end = err.ptr;

  /* Jeżeli znaleziono "OK", przejdź dalej,
   * jeśli naleziono "ERROR", zwróć błąd
   */
  if (resp->err != SIM_ok)
      return resp->err;

  /* Ustaw wskaźniki na zwrócone parametry */
  resp->err = SIM_retrieve(lines, "CREG", resp);

  /* Zwróć wynik ostatniej funkcji */
  return resp->err;
}
```

Każda funkcja callback przyjmuje kolejno:

- wskaźnik do tablicy struktur `SIM_line_pair`, reprezentujących linie znalezione w dopowiedzi,
- wskaźnik struktury `SIM_line_pair`, gdzie odnaleziono początek odpowiedzi na daną komendę,
- wskaźnik do struktury `SIM_resp`, przechowującej informacje o wyniku zapytania,
- wskaźnik void do struktury `SIM_intf`.

Wysłanie komendy jest następuje poprzez użycie funkcji `SIM_run(SIM_intf *, SIM_cmd *)`. Funkcja korzysta z mechanizmu *mutexów* (ang. *Mututual Exclusion*), dzięki czemu nawet jeśli dwa wątki będą próbowały wykonać funkcję, to nie dojdzie do naruszenia danych w razie próby dostępu do tego samego interfejsu UART lub struktury `SIM_intf`.

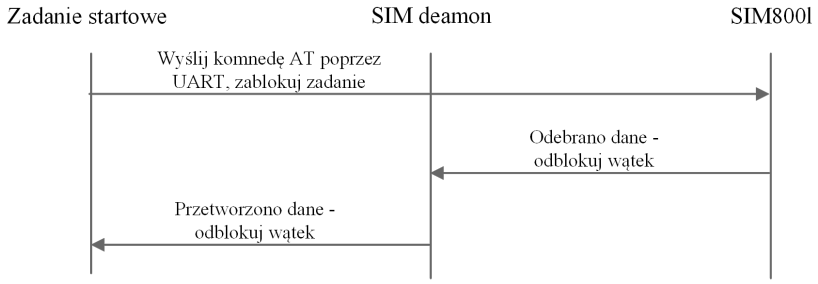
Funkcja blokuje dostęp poprzez mutex oraz wysyła łańcuch znaków wygenerowany w trakcie inicjalizacji. Następnie czeka aż SIM800 deamon przetworzy dane, blokując tym samym swój wątek. Blokowanie i odblokowywanie wątków jest wykonywane poprzez użycie mechanizmu kolejek (*Queues*) stanowiącego domyślny sposób komunikacji między wątkami w systemie *FreeRTOS*. Po odebraniu odpowiedzi, wątek jest odblokowywany poprzez wysłanie kodu wykonania do określonej kolejki. Funkcja zostaje odblokowana dla innych zadań i zwracany jest wynik operacji.

Listing 4: Funkcja `SIM_run`

```
SIM_error SIM_run(SIM_intf *sim, SIM_cmd *cmd)
{
  SIM_error err = SIM_ok;

  /* Zablokuj dostęp dla innych komend */
  xSemaphoreTake(sim->add_cmd_mutex, portMAX_DELAY);

  /* Wyślij komendę */
  if ((err = SIM_sendAT(sim, cmd->at)) == SIM_ok)
  {
      /* Poczekaj aż SIM800 deamon przetworzy dane */
```

```
        err = SIM_sub(sim, cmd);
    }

    /* Odblokuj dostęp dla innych komend */
    xSemaphoreGive(sim->add_cmd_mutex);

    return err;
}
```

Wykonanie powyższej funkcji dla komendy z jedną funkcją callback ilustruje rysunek 0.2.

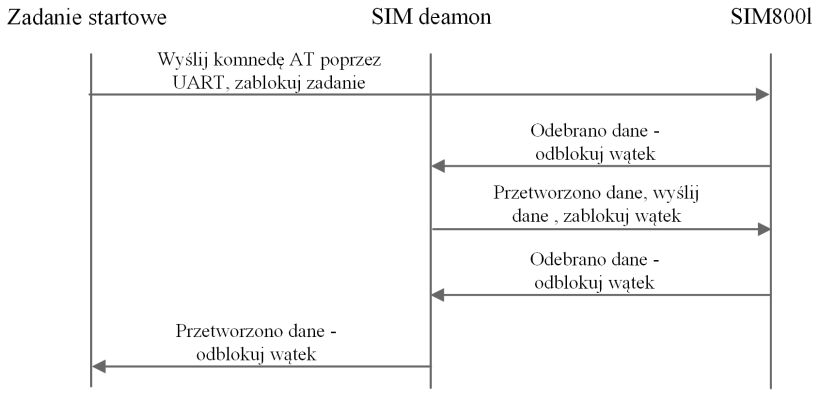


Rys. 0.2: Interakcja z wątkiem SIM800 deamon w przypadku użycia jednej funkcji *callback*

Wykorzystywane są również komendy, na które składa się kilka funkcji callback. Przykładowo jest to konieczne w przypadku komendy `CIPSEND`. Jak określa to dokumentacja, po wysłaniu komendy `AT+CIPSEND=<...>`:

1. Moduł wysyła znak ">", po otrzymaniu którego jednostka sterująca może dopiero przekazać dane przez UART.
2. Moduł zwraca wynik operacji (np. „`<n>, SEND OK`”, gdzie `<n>` to numer połączenia TCP).

Ponieważ operacje mogą być od siebie oddalone w czasie nawet o milisekundy, potrzebne jest wykorzystanie dwóch callbacków. Operacja taka jest przedstawiona na rysunku 0.3.



Rys. 0.3: Interakcja z wątkiem SIM800 deamon w przypadku użycia dwóch funkcji *callback*

0.0.1.4. Użycie biblioteki do komunikacji poprzez GPRS

Głównym powodem użycia dedykowanego wątku dla biblioteki jest komunikacja poprzez TCP, gdzie dane mogą zostać odebrane w każdej chwili od momentu połączenia z serwerem.

W celu korzystania z funkcji sieciowych, moduł musi zostać odpowiednio zainicjalizowany. Sposób inicjalizacji opisują jedna z not aplikacyjnych modułu [**?**], a jego użycie jest zaprezentowane w punkcie **??**.

Przed połączeniem wymagane jest wywołanie funkcji `SIM_error SIM_listenTCP(SIM_intf *sim, const SIM_con_num n, void (*resp_handler)(SIM_error *))`, inicjalizującej nowe połączenie w deamonie. Funkcja przyjmuje:

- `n` - numer połączenia (0 - 5),
- `(*resp_handler)(SIM_error *)` - wskaźnik do funkcji callback, która jest wywoływana w razie otrzymania danych lub zakończenia połączenia.

Funkcja wskazywana przez `*resp_handler` może wykonywać dowolną akcję. W opisywanym urządzeniu służy ona do odblokowywania zadania *MQTT deamon* poprzez użycie mechanizmu kolejek (*Queues*).

Inicjalizacja połączenia oznacza inicjalizację struktury `SIM_TCP_cmd`, która zawiera informacje o trwającym połączeniu.

Listing 5: Struktura `SIM_TCP_cmd`

```
typedef struct SIM_TCP_cmd
{
    SIM_con_num con;
    SIM_error (*handler)(SIM_line_pair *, SIM_line_pair *, SIM_resp
       ↪  *, void *);
    SIM_resp resp;
    void (*resp_handler)(SIM_error *err);
} SIM_TCP_cmd;
```

Aby połączyć się z serwerem potrzebne jest użycie funkcji `SIM_writeCIPSTART(SIM_cmd *cmd, const SIM_con_num n, char *mode, char *address, const unsigned int port)`. Funkcja przyjmuje następujące argumenty:

- `n` - numer połączenia (0 - 5),
- `mode` - tryb („`TCP`”) lub („`UDP`”),
- `address` - adres IP lub DNS serwera,
- `port` - numer portu.

Wysyłanie danych jest realizowane poprzez funkcję `SIM_data_len SIM_TCP_write(SIM_intf *sim, SIM_con_num n, void *buf, unsigned int len)` (będącej wrapperem funkcji `SIM_writeCIPSEND`), gdzie:

- `send_data` - wskaźnik do tablicy o dowolnym typie danych,
- `send_data_len` - ilość bajtów do wysłania.

Funkcja zwraca ilość wysłanych bajtów lub błąd jeśli jakiś wystąpił.

Zgodnie z rysunkiem 0.1, wątek *SIM800 deamon* wykrywa fazy powiązane z ruchem TCP, określone w dokumentacji techniczej [**?**]. Są one następujące:

1. „`+RECEIVE,<n>,<length>`”, oznaczające otrzymanie danych, gdzie „`<n>`” oznacza numer połączenia, „`<length>`” ilość otrzymanych bajtów. Po wysłaniu frazy, moduł wysyła wspomniane dane.
2. „`<n>,CLOSED`”, oznaczające zakończenie połączenia przez serwer, gdzie „`<n>`” oznacza numer połączenia.

Po otrzymaniu którejś z tych odpowiedzi, deamon uruchamia callback określony wcześniej podczas wywołania `SIM_listenTCP`.

W przypadku otrzymania `+RECEIVE,<n>,<length>`, deamon musi zapisać dane otrzymane od modułu w strukturze `SIM_resp`, mieszczącej się w strukturze `SIM_TCP_cmd`. Nadpisywanie tych danych oraz ich odczyt są wzajemnie blokowane przy użyciu mutexu, dzięki czemu można zapobiec wszelkim wyścigom danych.

Odczyt odebranych danych powinien być wykonywany przy użyciu `SIM_data_len SIM_TCP_read(SIM_intf *sim, SIM_con_num n, void *buf, unsigned int len)`, gdzie:

- `n` - numer połączenia (0 - 5),
- `buf` - bufor, do którego skopiowane będą dane,
- `len` - maksymalna ilość bajtów do skopiowania.

Funkcja zwraca ilość bajtów, które zostały skopiowane.