Projektowanie urządzeń z modułami GSM (3)

Oprogramowanie mikrokontrolera

W trzeciej części cyklu poświęconego projektowaniu urządzeń z modułami GSM zajmiemy się zagadnieniem projektowania oprogramowania mikrokontrolera współpracującego z modułem GSM.

Zadania stojące przed projektantem oprogramowania, to:

- Włączanie modułu i przesyłanie poleceń inicjujących.
- Sprawdzanie poprawności pracy modułu i restartowanie go w razie potrzeby.
- Komunikacja z modułem w celu zapewnienia założonej funkcjonalności urządzenia.

W dokumentacji większości modułów GSM znajdziemy uwagę o możliwości zawieszenia modułu i konieczności wykrywania stanu zawieszenia i restartu modułu podczas pracy urządzenia. Sama komunikacja z modułem musi odbywać się z zachowaniem zasad podanych w dokumentacji. Należy zwrócić uwagę m.in. na:

- Możliwość przesyłania poleceń i danych wyłącznie wtedy, gdy moduł jest na to gotowy (aktywny stan linii CTS).
- Wymagane odstępy czasowe pomiędzy poleceniami oraz pomiędzy odpowiedzią modułu na polecenie i kolejnym poleceniem.
- Zachowanie odpowiedniej sekwencji zdarzeń przy poleceniach złożonych – dane można przesyłać po zgłoszeniu gotowości modułu.
- Możliwość przesłania przez moduł komunikatu asynchronicznego, niezależnego od przesyłanych do modułu poleceń.

Błędem często popełnianym przez początkującym programistów jest projektowanie oprogramowania do komunikacji z modułem w postaci sekwencji przeplatających się akcji przesłania polecenia i oczekiwania na odpowiedź. Odpowiedź z modułu może nigdy nie nadejść lub moduł może przesłać komunikat niebędący odpowiedzi na przesłane wcześniej polecenie. Ponadto struktura programu musi przewidywać restart i powtórne inicjowanie modułu w dowolnym kontekście komunikacji.

Właściwym podejściem do konstrukcji oprogramowania jest napisanie go w postaci kilku współpracujących ze sobą automatów. Istotne jest rozdzielenie funkcji nadawania poleceń i odbioru odpowiedzi modułu. Nie należy zbyt silnie wiązać w oprogramowaniu obu tych czynności. Automat zajmujący się odbiorem danych z modułu musi być gotowy do odbioru dowolnego komunikatu, a nie tylko spodziewanej odpowiedzi na polecenie. Automat wysyłający polecenia musi uwzględniać limity czasu odpowiedzi i zapewniać stosowną reakcję na brak odpowiedzi. Przy braku aktywności urządzenia trzeba okresowo sprawdzać poprawność działania modułu, np. przez przesyłanie pustego polecenia AT.

Przykładowa aplikacja

W celu zilustrowania procesu projektowania programu zdefiniujemy przykładową aplikację, w której komunikacja przez sieć GSM polega na wymianie wiadomości SMS. Model urządzenia został wykonany przy użyciu płytki STM32F0DISCOVERY z mikrokontrolerem STM32F051 i płytki z modułem Quectel M95. Do płytki DISCOVERY dołączono wyświetlacz LCD i diodę LED RGB.

Przy wykryciu zdarzenia (naciśnięcia przycisku) urządzenie wysyła wiadomość z informacją o napięciu zasilania i temperaturze. Przychodzące wiadomości SMS mogą sterować świeceniem diody RGB poprzez ustawienie jednego z ośmiu stopni jasności każdej ze składowych.

Oprogramowanie musi zapewniać:

- Inicjowanie modułu GSM oraz jego restart w wypadku zawieszenia.
- Wysyłanie wiadomości SMS.
- Odbiór i rozpoznawanie treści wiadomości SMS.

Cały projekt oprogramowania znajduje się w pliku M95_sms_demo.zip. W artykule pokazano jedynie najistotniejsze fragmenty programu związane z obsługą komunikacji z modułem GSM.

Odbiór i rozpoznawanie komunikatów z modułu – parser odpowiedzi

Projekt oprogramowania zaczniemy od części odpowiedzialnej za odbiór komunikatów z modułu. Parser komunikatów ma za zadanie wykrywanie odpowiedzi modułu GSM na istotne polecenia używane w aplikacji oraz wyłuskiwanie z odpowiedzi potrzebnych danych, np. numerów telefonów inicjujących połączenia lub treści SMS. Parser nie musi identyfikować wszystkich odpowiedzi modułu, lecz jedynie te, których odbiór jest istotny w danym zastosowaniu. W naszym przypadku będą to:

- Echo pustego polecenia AT.
- Odpowiedź OK kończąca poprawne wykonanie polecenia.
- Odpowiedź +CMGL: zawierająca treść wiadomości SMS.

Spośród tych trzech komunikatów jedynie +CMTI wymaga interpretacji treści. Pozostałe trzy muszą być jedynie wykrywane. Rozpoznanie każdego z wymienionych komunikatów powoduje ustawienie odpowiadającego mu znacznika.

Informacja o odebranej wiadomości ma postać
+CMGL: 1,"REC READ","+48600123456","","2015/05/29 12:49:50+08"
po czym, w następnym wierszu, następuje treść wiadomości.

Parsowanie komunikatów z modułu GSM nie wymaga buforowania całej ich treści; może ono następować znak po znaku, w miarę odbierania kolejnych bajtów przez interfejs UART. Zapisu do pamięci wymagają tylko dane z przychodzącej wiadomości – numer wiadomości (liczba następująca po znaku dwukropka), numer telefonu nadawcy i treść wiadomości.

Procedura parsująca dane z modułu GSM ma postać prostego automatu. Jest ona wywoływana przy odbiorze znaków z modemu następującym w przerwaniu z modułu UART. W przykładowym programie procedura ta nosi nazwę iparse(). Zrealizowany w niej automat ma 9 stanów zdefiniowanych przez typ wyliczeniowy istate_, z czego 4 są związane z analizą treści przychodzącej wiadomości SMS.

Do identyfikacji komunikatów procedura posługuje się tablicą wzorców początków komunikatów *msg[]*, posortowaną alfabetycznie. Tablicy tej odpowiada typ wyliczeniowy *imsg_* definiujacy symboliczne identyfikatory komunikatów.

Stanem początkowym parsera jest stan *IS_START*. Odebranie pierwszego znaku różnego od spacji i zgodnego z jednym z rozpoznawanych komunikatów powoduje przejście do stanu rozpoznawania komunikatu *IS_MATCHING*. Jeżeli pierwszy lub kolejny znak nie pasuje do żadnego z definiowanych wzorców, następuje przejście do stanu oczekiwania na początek wiersza *IS_SKIPEOLN*.

Każda dłuższa przerwa w odbiorze danych powoduje przejście automatu parsera do stanu początkowego.

Stwierdzenie zgodności z całym wzorcem komunikatu powoduje ustawienie znacznika odbioru danego komunikatu i przejście do oczekiwania na początek wiersza. Jedynie komunikat z treścią odebranej wiadomości wymaga bardziej złożonej obsługi- W tym przypadku automat przechodzi kolejno przez stany:

- Oczekiwania na numer wiadomości *IS_GETMSG1*.
- Gromadzenia numeru wiadomości *IS_GETMSGNUM*.
- Szukania numeru telefonu nadawcy *IS_MSGFINDNUM*.
- Gromadzenia numeru nadawcy *IS_MSGGETNUM*.
- Pomijania dalszej części komunikatu *IS_MSGSKIPEOLN*.
- Gromadzenia treści wiadomości *IS_MSGBODY*.

Zakończenie odczytywania wiadomości powoduje ustawienie znacznika *msg_rdy*. Numer wewnętrzny wiadomości, numer telefonu nadawcy i treść wiadomości zostają umieszczone odpowiednio w zmiennych łańcuchowych *msgnum*, *msgphnum* i *msgbody*.

Transmisja poleceń do modułu GSM

Transmisja danych przez moduł UART do modemu GSM jest realizowana w procedurze obsługi przerwania UART. Ponieważ polecenia są przesyłane pojedynczo, nie jest potrzebne ich dodatkowe buforowania w kolejce FIFO. Procedura transmisji łańcucha, z której korzysta główny automat, wpisuje jedynie adres początkowy polecenia do zmiennej *sendptr* i odblokowuje przerwanie nadajnika.

Struktura oprogramowania urządzenia

Całe oprogramowanie składa się z procedury inicjującej oraz dwóch procedur obsługi przerwań: timera SysTick i UART. Procedura inicjująca korzysta ze struktury danych zawierającej adresy inicjowanych rejestrów i ich wartości. Przerwanie Systick jest wywoływane z częstotliwością 1600 Hz, wynikającą z zastosowanej metody obsługi wyświetlacza LCD. Obsługa pomiarów ADC i przycisków oraz automat główny są wywołane z częstotliwością 100 Hz. Oprogramowanie naprzemiennie mierzy i filtruje wartości napięcia zasilania i temperatury urządzenia.

Działanie fragmentów oprogramowania niezwiązanych z obsługą modułu GSM zostało opisane w serii artykułów „32 bity jak najprościej", prezentowanej niedawno w EP.

Główny automat

Główny automat urządzenia odpowiada za współpracę z modułem GSM i realizację podstawowej funkcjonalności urządzenia. Logika automatu została zawarta w procedurze *M95_handler()*, wywoływanej z procedury obsługi przerwania timera systemowego. Zmiany stanów automatu zachodzą na podstawie zdarzeń specyficznych dla poszczególnych stanów oraz – w przypadku niewystąpienia zdarzenia – w wyniku przekroczenia limitu czasu spędzanego w poszczególnych stanach (**listing 1**).

Ponieważ każdy stan ma określony limit czasu, a reakcja na zdarzenia następuje tylko w niektórych stanach, wygodniej było zrealizować automat w niezbyt typowej postaci nie jednej, a dwóch instrukcji *switch()*. Pierwsza instrukcja *switch()* zapewnia obsługę przekroczenia limitów czasu zależną od stanu automatu. Druga instrukcja *switch()* obsługuje wyłącznie zmiany stanów wynikające z innych zdarzeń.

Stan automatu jest przechowywany w zmiennej typu wyliczeniowego *mstate*.

Stanem początkowym jest stan *MS_INIT*, z którego wyjście następuje wyłączeni w wyniku upłynięcia limitu czasu. Przy starcie oprogramowania limit jest ustawiana na 3 sekundy, co zapewnia odczekanie odpowiedniego czasu dla stabilizacji zasilania całego urządzenia. Po tym czasie następuje rozpoczęcie włączania modułu GSM i przejście do stanu włączania *MS_TURNON*, w którym generowany jest impuls włączający moduł o wymaganym czasie trwania 2,2 s. Po tym czasie następuje zakończenia impulsu włączającego i przejście do stanu *MS_SYNC*. W stanie *MS_SYNC*, po upływie czasu 0,5 sekundy, jest przesyłane puste polecenie AT służące do synchronizacji interfejsu UART modułu, po czym następuje przejście do stanu *MS_SYNC2*.

Odebranie odpowiedzi AT<CR><LF>OK<CR<LF> w stanie MS_SYNC2 powoduje wysłanie do modułu polecenia AT+CMGF=1, ustawiającego tekstową reprezentację wiadomości SMS. Gdyby do zainicjowania modułu były potrzebne inne polecenia (np. wprowadzenia PIN), należałoby je analogicznie obudować kolejnymi stanami

Listing 1. Procedura obsługi przerwania UART i rozpoznawania komunikatów modułu GSM

```
static uint16_t itimout = 0;
static _Bool at_received = 0, ok_received = 0;
static _Bool msg_rdy = 0;
static _Bool send_report = 0;

static uint8_t msg_num;
#define MSGSIZE   16
static char msgnum[3], msgbody[MSGSIZE];
static char msgphnum[16];
//=========================================================
enum istate_ {IS_START, IS_SKIPEOLN, IS_MATCHING,
    IS_MSGBODY, IS_GETMSG1, IS_GETMSGNUM, IS_MSGFINDNUM,
    IS_MSGGETNUM,
    IS_MSGSKIPEOLN};

// incoming message identifiers
enum imsg_ {IM_PCMGL, IM_PCMTI, IM_AT, IM_OK, IM_RING};
static const char * const msg[] = {
    „+CMGL:”,
    „AT\r”,
    „OK”,
    „RING”,
    0
};

//=========================================================
void iparse(uint32_t c)
{
    static enum istate_ istate = IS_START;
    static enum imsg_ imsgidx;
    static uint8_t imsgpos;
    static uint8_t idx, mi;
    if (itimout == 0)
        istate = IS_START;  // reset state if timeout
    itimout = 100;
// force uppercase
    if (istate != IS_MSGBODY && c >= ‚a’ && c <= ‚z’)
        c -= ‚a’ - ‚A’;
    switch (istate)
    {
        case IS_SKIPEOLN:
            if (c == ‚\n’)
            istate = IS_START;
            break;
        case IS_START:  // 1st char in a line
            if (c > ‚ ‚)
            {
                for (imsgidx = IM_PCMGL; msg[imsgidx] && c > msg[imsgidx][0]; imsgidx ++);
                if (c == msg[imsgidx][0])
                {
                    imsgpos = 1;
                    istate = IS_MATCHING;
                    break;
                }
            }
            istate = IS_SKIPEOLN;
            break;
        case IS_MATCHING:
            while (msg[imsgidx] && c > msg[imsgidx][imsgpos] && strncmp(msg[imsgidx], msg[imsgidx +
1], imsgpos))
            imsgidx ++;
            if (msg[imsgidx] && c == msg[imsgidx][imsgpos])
            {
                if (!msg[imsgidx][++ imsgpos])
                {
                // Matched!
                    istate = IS_SKIPEOLN;
                    switch (imsgidx)
                    {
                        case IM_AT:
                            at_received = 1;
                            break;
                        case IM_PCMGL:
                            if (!msg_rdy) istate = IS_GETMSG1;
                            break;
                        case IM_OK:
                            ok_received = 1;
                            break;
                        case IM_RING:
                            memcpy(lcd.screen[1], „Ring!  „, 7);
                            dis_blue_target = LED_MAX;
                            blue_timer = 500;
                            break;
                    }
                    lcd.req.upd[1] = 1;
                    blt_on();
                }
            }
            else istate = IS_SKIPEOLN;
            break;
        case IS_GETMSG1:
// Format: +CMGL: 1,”REC READ”,”+48600123456”,””,”2014/05/29 12:49:50+08”
            if (c == ‚\n’)
                istate = IS_START;
                else if (c >= ‚0’ && c <= ‚9’)
                {
                // internal message number
                    msgnum[0] = c;
                    mi = 1;
                    msg_num = c - ‚0’;
                    istate = IS_GETMSGNUM;
```

składającymi się na sekwencję inicjującą.

Przekroczenie limitu czasu odpowiedzi na którekolwiek polecenie wysyłane do modułu powoduje restart modułu, którego algorytm jest opisany w dalszej części artykułu.

Pomyślne zakończenie inicjowania modułu, sygnalizowane odpowiedzią OK na ostatnie polecenie inicjujące powoduje przejście do stanu *MS_RDY*. W stanie tym automat sprawdza, czy wystąpiło zdarzenie wymagające przesłania SMS i w razie takiej potrzeby wysyła do modułu polecenie inicjujące przesyłania wiadomości w postaci AT+CMGS=”<nr_teL_odbiorcy>”<CR>, po czym przechodzi do stanu *MS_SENDING*, w którym oczekuje na zgłoszenie gotowości modułu do przekazania treści wiadomości.

Jeżeli w stanie *MS_RDY* w ciągu 10 sekund nie nastąpi zdarzenie, do modułu jest przesyłane polecenie AT+CMGL=”ALL”<CR>, które powoduje przesłanie przez moduł listy wszystkich odebranych wiadomości, po czym automat przechodzi do stanu *MS_PING*. Gdyby nasze urządzenie nie reagowało na SMS, należałoby w tym przypadku przesyłać dowolne inne polecenie, np. puste, w celu okresowego testu aktywności modułu. Brak odpowiedzi na to polecenie powinien powodować restart modułu.

Odpowiedź na polecenie AT+CMGL zawiera listę wiadomości, zakończoną napisem OK. Każdy element listy jest rozpoznawany przez parser odpowiedzi z modułu, który zapisuje dane pierwszej wiadomości z listy. W ten sposób po zakończeniu odpowiedzi, co jest sygnalizowane przez parser ustawieniem znacznika *ok_received* i pobraniu przez parser wiadomości (znacznik *msg_rdy*) automat może zinterpretować treść pierwszej wiadomości, a następnie wydać polecenie jej skasowania AT+CMGD=<numer_wiadomości>, niezbędne dla umożliwienia odbioru kolejnych wiadomości (**listing 2**).

Restart modułu

Restart modułu jest inicjowany w przypadku niewykrycia

Listing 1. c.d.

```
                }
                else if (c != , ,) istate = IS_SKIPEOLN;
                break;
        case IS_GETMSGNUM:  // collect message number
            if (c >= ,0' && c <= ,9')
            {
                msgnum[mi ++] = c;
                msg_num *= 10;
                msg_num += c - ,0';
            }
            else if (c == ,,')
            {
                msgnum[mi] = 0;
                mi = 3;  // skip 3 „
                istate = IS_MSGFINDNUM;
            }
            else istate = IS_MSGSKIPEOLN;
            break;
        case IS_MSGFINDNUM:  // find sender's phone number
            if (c == ,"')
            {
                lcd.screen[1][mi] = ,"';
                if (-- mi == 0)
                {
                    idx = 0;
                    istate = IS_MSGGETNUM;
                }
            }
            else if (c < , ,) istate = IS_SKIPEOLN;
            break;
        case IS_MSGGETNUM:  // collect sender's phone numer
            if (c != ,"')
            {
                lcd.screen[0][idx ++] = c;
                if (mi < 16) msgphnum[mi ++] = c;
            }
            else
            {
                msgphnum[mi] = 0;
                while (idx < 16) lcd.screen[0][idx ++] = , ,;
                istate = IS_MSGSKIPEOLN;
            }
            break;
        case IS_MSGSKIPEOLN:
            if (c == ,\n')
            {
                idx = 0;
                mi = 0;
                ok_received = 0;
                istate = IS_MSGBODY;
            }
            break;
        case IS_MSGBODY:  // copy message text
            if (c >= , ,)
            {
                if (mi < MSGSIZE - 1) msgbody[mi ++] = c;
                // Message procesing
                if (idx < 16) lcd.screen[1][idx ++] = c;
            }
            else
            {
                // end of message
                msgbody[mi] = 0;
                while (idx < 16) lcd.screen[1][idx ++] = , ,;
                lcd.req.upd[0] = 1;
                lcd.req.upd[1] = 1;
                blt_on();
                msg_rdy = 1;
                istate = IS_START;
            }
            break;
    }
}

// send a string using UART interrupt
static const char *sendptr = 0;
static void send_string(const char * s)
{
    sendptr = s;
    USART1->CR1 |= USART_CR1_TXEIE | USART_CR1_RE;
}

void USART1_IRQHandler(void)
{
    if (USART1->ISR & USART_ISR_RXNE)  // data received
    {
        uint32_t c;
        c = USART1->RDR;
        iparse(c);
    }
    if (USART1->ISR & USART1->CR1 & USART_ISR_TXE)
    {
        USART1->TDR = *sendptr ++;
        if (*sendptr == 0) USART1->CR1 &= ~USART_CR1_TXEIE;
    }
}
```

Listing 1: Główny automat obsługi modułu M95 i obsługa przerwani SysTick.

```
// M95 stuff
```

Listing 1. c.d.

```
#define M95_PWRON_TIME     220
#define M95_PWROFF_TIME    80
#define M95_EMGOFF_TIME    40

static char cmgd[] = „AT+CMGD=xx\r”;

static char status[18] = „                \x1a”;


static void m95_handler(void)
{
    static enum {MS_INIT, MS_TURNON, MS_SYNC, MS_SYNC2,
        MS_INITCMD,
        MS_TURNOFF, MS_EMGOFF, MS_RDY, MS_PING,
        MS_SENDING, MS_DELMSG} mstate = MS_INIT;
    static uint16_t tout = 300;
    static _Bool restarting = 0;
    if (tout && -- tout == 0)
    {
        // change state after timeout
        switch (mstate)
        {
        case MS_INIT:  // start ON pulse
            M95_PWR_PORT->BSRR = 1 << M95_PWR_BIT;
            tout = M95_PWRON_TIME;
            memcpy(lcd.screen[1], „GSM On          „, 14);
            lcd.req.upd[1] = 1;
            blt_on();
            mstate = MS_TURNON;
            break;
        case MS_TURNON:  // end ON pulse
            M95_PWR_PORT->BRR = 1 << M95_PWR_BIT;
            tout = 500;
            mstate = MS_SYNC;
            break;
        case MS_SYNC:     // send initial cmd
            at_received = 0;
            ok_received = 0;
            send_string(„AT\r”);
            tout = 100;
            mstate = MS_SYNC2;
            break;
        case MS_TURNOFF:  // end OFF pulse
            M95_PWR_PORT->BRR = 1 << M95_PWR_BIT;
            M95_EMGOFF_PORT->BRR = 1 << M95_EMGOFF_BIT;
            tout = 300;
            mstate = MS_INIT;
            break;
        case MS_RDY:  // ping GSM module
            at_received = 0;
            ok_received = 0;
            send_string(„AT+CMGL=\”ALL\”\r”);
            dis_green_target = LED_MAX;
            green_timer = 100;
            tout = 100;
            mstate = MS_PING;
            break;
        case MS_SENDING:  // send message body
            send_string(status);
            tout = 100;
            mstate = MS_RDY;
            break;
        default: // other timeout - restart
            if (restarting)
            {
                // already restarting - emergency off
                M95_EMGOFF_PORT->BSRR = 1 << M95_EMGOFF_BIT;
            }
            else
            {
                // normal turnoff
                restarting = 1;
                M95_PWR_PORT->BSRR = 1 << M95_PWR_BIT;
            }
            tout = M95_PWROFF_TIME;
            mstate = MS_TURNOFF;
        }
    }
    else
    {
        // state changes due to events
        switch (mstate)
        {
            case MS_SYNC2:
                if (at_received && ok_received)
                {
                    ok_received = 0;
                    send_string(„AT+CMGF=1\r”);
                    tout = 200;
                    mstate = MS_INITCMD;
                }
                break;
            case MS_PING:
                if (ok_received)
                {
                    ok_received = 0;
                    if (msg_rdy)
                    {
                        uint32_t i = 8;
                        // process message
                        // „Lrgb” message controls rgb LED
```

odpowiedzi modułu w określonym indywidualnie dla poszczególnych poleceń maksymalnym czasie odpowiedzi. Moduł M95 jest wyposażony przez producenta w dwa mechanizmy restartu: zwykły, poprzez normalne wyłączenie modułu po wcześniejszym wylogowaniu z sieci oraz awaryjny, wymuszający wyłączenie modułu nawet przy całkowitym zawieszeniu oprogramowania. Przekroczenie limitu czasu odpowiedz na polecenie powoduje uruchomienie jednego z tych mechanizmów. Początkowo jest to mechanizm normalnego wyłączenia poprzez podanie na wejście włączające impulsu o czasie 800 ms. Jeżeli po próbie powtórnego włączenia moduł nie odpowiada na polecenia, zostaje użyty mechanizm restartu awaryjnego, wyzwalany poprzez podanie na wejście EMG_OFF impulsu o czasie 400 ms. Oby typy wyłączenia są obsługiwane w jednym stanie MS_TURNOFF, po którym następuje przejście do stanu MS_INIT, w którym po 3 sekundach nastąpi włączenie modułu.

Podsumowanie

Opisana struktura oprogramowanie zapewnia poprawną współpracę urządzenia mikroprocesorowego z modułem GSM Quectel M95. Analogiczny schemat może zostać zastosowany dla dowolnego innego modułu, po odpowiedniej modyfikacji parametrów czasowych i sekwencji inicjującej, co może wiązać się ze zwiększeniem liczby stanów automatu. Funkcjonalność urządzenia w zakresie współpracy z modułem GSM może zostać łatwo rozbudowana poprzez dodanie kolejnych odpowiedzi modemu do listy odpowiedzi rozpoznawanych przez parser. I ewentualną rozbudowę parsera w celu zapewnienia rozpoznawania elementów treści tych odpowiedzi. Oprogramowanie zbudowane w taki sposób zapewnia pełny asynchronizm i nie wymaga oczekiwania w pętli zdarzeń programu na zdarzenie, które może nigdy nie wystąpić, dzięki czemu można łatwo zaimplementować obsługę sytuacji awaryjnych.

Grzegorz Mazur

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

Listing 1. c.d.

```
                        // r, g, b - digits 0..7
                        if (msgbody[0] == ,L')
                        {
                            target.red = (msgbody[1] & 7) << 3;
                            target.green = (msgbody[2] & 7) << 3;
                            target.blue = (msgbody[3] & 7) << 3;
                        }
                        // prepare delete command
                        cmgd[i ++] = msgnum[0];
                        if (msgnum[1]) cmgd[i ++] = msgnum[1];
                        cmgd[i ++] = ,\r';
                        cmgd[i] = 0;
                        lcd.req.upd[0] = 1;
                        send_string(cmgd);
                        msg_rdy = 0;
                    }
                    tout = 1000;
                    mstate = MS_RDY;
                }
                break;
            case MS_INITCMD:
                if (ok_received)
                {
                    ok_received = 0;
                    tout = 1000;
                    restarting = 0;
                    mstate = MS_RDY;
                }
                break;
            case MS_RDY:
                if (send_report)
                {
                    send_report = 0;
#error „edit phone number below, then disable this message"
                    send_string(„AT+CMGS=\"+48600123456\"\r");
                    tout = 100;
                    mstate = MS_SENDING;
                }
                break;
            default:
                ;
        }
    }
    // show state
    lcd.screen[1][15] = mstate + ,0';
    lcd.req.upd[1] = 1;
}

void SysTick_Handler(void)
{
    static uint8_t tdiv = 0;
    static uint8_t sdiv = 0;
    static uint32_t adc_avg[2];
    static uint8_t dkeystate = 0, keylstate = 0, keycstate = 0, keyrstate = 0;
    lcdhandler();
    if ((++ tdiv & 15) == 0)
    {
        // 10 ms interrupt
        static enum {ADCH_TS, ADCH_VREF} adch = ADCH_TS;
        m95_handler();
        if (itimout) -- itimout;
        // Discovery board button
        if ((dkeystate = dkeystate << 1 | (BUTTON_PORT->IDR >> BUTTON_BIT & 1)) == 1)
        {
            send_report = 1;
        }
        // ADC - temperature and supply voltage measurements
        if (ADC1->ISR & ADC_ISR_EOC)
        {
            uint32_t val = ADC1->DR;       // 0 before first conversion
            if (adc_avg[adch] == 0)
            {
                // initial measure - set
                adc_avg[adch] = val << AVG_SHIFT;
            }
            else
            {
                // low-pass filters
                adc_avg[adch] = adc_avg[adch] + val - (adc_avg[adch] >> AVG_SHIFT);
            }
            if (++ adch > ADCH_VREF) adch = ADCH_TS;
        }
        else if (ADC1->ISR & ADC_ISR_ADRDY)
        {
            // ready for conversion
            ADC1->CR = ADC_CR_ADSTART | ADC_CR_ADEN;        // start cont. conversion
        }
        else if ((ADC1->CR & (ADC_CR_ADCAL | ADC_CR_ADEN)) == 0)
        {
            // calibrated but not enabled yet - enable
            ADC1->CR = ADC_CR_ADEN;
        }
    }
}
```