APPLICATION NOTE

OM4754
EAMPS Software User Guide

AN95030

Philips
Semiconductors

PHILIPS

Abstract

This document provides details of the cellular radio software developed by Product Concept and Application Laboratory in Eindhoven (PCALE). This software is demonstrated on the OM4753C AMPS demonstration and emulation unit and implements the EIA/TIA-553 standard.

APPLICATION NOTE

OM4754
EAMPS Software User Guide

AN95030

Author(s):

R.S.M.J. Kempen
N. Barendse
Product Concept & Application Laboratory Eindhoven,
The Netherlands

Keywords

Cellular radio
Cellular Chipset
C-code Software

Date: 1 May, 1995

Summary

This document describes the C-code software used in the Philips Cellular Radio Demonstration Unit, detailing the overall structure of the software, along with the functionality of each sub-module used.

In addition, information is given on how to set up the RTX-51 operating system and how to use the KEIL/Franklin compiler.

The chipset used in the AMPS cellular demoboard consists of:

- 83CL580 Micro Controller
- PSD312L Programmable Micro Controller Peripheral
- UMA1000LT Data Processor (DPROC)
- SA5752 and SA5753 Audio Processors (APROC)
- TDA7050 Audio Amplifier
- UMA1015M Dual synthesizer
- ST25C02A 256 bytes EEPROM
- LP3800-A LCD display.

CONTENTS

1. Introduction

1.1 Purpose

This document provides details of the cellular radio software developed by Product Concept and Application Laboratory in Eindhoven (PCALE). This software is demonstrated on the OM4753C AMPS demonstration and emulation unit and implements the EIA/TIA-553 standard[1].

1.2 Scope

This document describes how the software is implemented, details about the programming language (C) and the AMPS protocol[1] are not given.

1.3 Abbreviations

ADC	Analog to Digital Converter
AMPS	Advanced Mobile Phone Service
APROC	Audio Processor
DPROC	Data Processor
EAMPS	Extended Advanced Mobile Phone Service
EEPROM	Electrically Erasable Programmable Read Only Memory
ETACS	Extended Total Access Communications System
I/O	Input/Output
LCD	Liquid Crystal Display
MMI	Man Machine Interface
PCALE	Product Concept and Application Laboratory in Eindhoven
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read Only Memory
SAT	Supervisory Audio Tone
SCC	Sat Colour Code
TACS	Total Access Communications System
VCO	Voltage Controlled Oscillator
VOX	Voice Operated transmission

1.4 References

[1] EIA/TIA-553 Mobile station- Land station compatibility specification, September 1989

[2] ETT/UM95002.0, OM4753C User Manual EAMPS demonstration and emulation unit, May 1, 1995

[3] C51 Compiler, C-Compiler, Run-Time-Library User’s guide 11.93

[4] RTX-51 Realtime Multitasking Executive for the 8051 Micro controller family User’s guide 10.91

1.5 Applicable documents

- SA5752, Audio processor - companding, VOX and amplifier, Product Specification, December 6, 1993.
- SA5753, Audio processor - filter and control section, Product Specification, December 6, 1993.
- UMA1000LT, Data processor for cellular radio (DPROC), objective specification, June 1993.
- P83CL580, Low Voltage single chip 8-bit micro controller, objective specification, September 1993.
- TDA7050, Low Voltage mono/stereo audio amplifier, data sheet, March 1991.
- OM5300, AMPS/TACS hybrid base band module BBM-2, objective specification, March 1995.
- UMA1015M, Low power dual frequency synthesizer for radio communications, objective specifications, March 1994.
- PSD312L, 3-volt single chip microcontroller peripheral, objective specification, May 1993.
- SA606, Low Voltage high performance mixer FM-IF system, product specification, November 1993.
- SA601, Low voltage LNA and mixer, 1 GHz, objective specification, December 1993.
- AN1741, Using the NE5750 and NE5751 for audio processing, Application Note, May 29, 1991.
- AN1742, Using the APROCII for low voltage design, Application Note, June 28, 1993.
- ETT/AN93016, UMA1015M Low Power Dual 1 GHz Frequency Synthesizer, Application Note, November 18, 1993.

1.6 Organization of this document

Chapter 2 contains a description of how to generate an executable system. It provides information about how to use the assembler, compiler and linker. It also gives information about how to configure the operating system and the PSD312L device.

Chapter 3 contains a description about the usage of the software. In chapter 4 one can find a general description about the operating system together with the specific implementations used in the software.

Chapter 5 contains a complete description of the software. In section 1 all tasks are explained. In section 2 the messages sent via mailboxes are described. In section 3 the sequence of messages sent via the mailboxes is described, and in section 4 the drivers are explained.

2. Generating an executable system

To generate an executable system the batch file `genall.bat` has to be executed. This batch file calls several other batch files. A list of all batch files used to generate an executable system is given below:

`assemble.bat`	To assemble a given file.
`compile.bat`	To compile a specific C-file, the extension .c must be omitted.
`genall.bat`	To generate an executable system.
`link.bat`	To link all objects to one executable system.

The file `amps.lnk` contains a link script for the application. Before generating an executable system the next paragraphs should be read.

2.1 Requirements

To be able to generate an executable system the following software must be installed on your local disk:

- KEIL/Franklin RTX-51, Real-Time Multitasking Executive

- KEIL/Franklin BL51, Banked Linker Locator
- KEIL/Franklin C51 Compiler, Standard Edition
- KEIL/Franklin A51 Assembler, 8051 Macro Assembler
- PSD-Gold/PSD-Silver Development System

Details on the installation of the KEIL/Franklin or PSD-Gold/PSD-Silver packages are not given in this document for these are included in the manuals which accompany these packages. It is therefore assumed in the following description of the set-up procedure that the packages have been installed.

For more details about the Macro assembler, C compiler, Object linker, Translation utilities and the PSD-Gold/ PSD-Silver Development system please consult the appropriate manuals.

2.2 Macro Assembler

In order to assemble the assembler file `example.asm` the (marco) assembler should be called like:

```
a51 example.asm NOMOD51
```

The assembler control `NOMOD51` causes all 8051 symbols to be unknown to the assembler. This allows the user to define definition files for other processors in the 8051 family (e.g. 83CL580). The definition file can be included using the `INCLUDE` control. The definition file for this application is called `reg580.inc` and must be included in all assembler files.

To assemble the above example the batch file `assemble.bat` can also be used like:

```
assemble example.asm
```

2.3 C compiler

In order to compile the C file `example.c` the compiler should be called like:

```
c51 example.c code objectextend define(PRODUCTION_PHONE) large symbols rom(large)
```

The compiler directives used are explained below:

`code`	Appends an assembly mnemonics list to the listing file.
`objectextend`	The generated code will contain additional information about variables.
`define(PRODUCTION_PHONE)`	Set’s the compiler switch `PRODUCTION_PHONE`.
`large`	Selects the `LARGE` memory model.
`symbols`	Generates a list of symbols used in and by the module being compiled.
`rom(large)`	Forces the `CALL` and `JMP` instructions to be coded as `LCALL` and `LJMP`.

The compiler switch `PRODUCTION_PHONE` is used to enable the IDLE mode of the micro controller and to disable the I^2C bus for other masters than the micro controller. For further details about the IDLE mode and how to disable the I^2C bus please refer to the chapters about the Idle task and the I^2C driver respectively.

To compile the above example the batch file compile.bat can also be used like:

```
compile example
```

In addition the compiler directive `debug` can be used to generate code for debugging.

2.4 Object Linker

In order to link an application from a link script `example.lnk` the linker should be called like:

```
bl51 @example.lnk
```

The file `example.lnk` is shown below:

```
cstartup.obj,                /* Initialization of RAM at startup   */
rtxconf.obj,                 /* RTX-51 configuration file          */
example.obj                  /* A simple example program           */
to example.abs               /* Output file-name                   */
map                          /* Generate memory map-file           */
nooverlay                    /* No Overlay on local segments       */
publics                      /* Public symbols in map-file         */
symbols                      /* Local symbols in map-file          */
ramsize(256)                 /* Set 83CL580 on-chip RAM size       */
rtx51                        /* Use RTX-51 operating system        */
```

The example above links the three object files `cstartup.obj`, `rtxconf.obj` and `example.obj` to an absolute file `example.abs` using the RTX-51 operating system.

2.5 Translation utilities

In order to convert an absolute file `example.abs` to an Intel hex file `example.hex` the translator utility oh51 should be called like:

```
oh51 example.abs
```

The output file `example.hex` can now be used to be programmed in the PSD312L device.

2.6 Configurating RTX-51

To configure the RTX-51 Operating system the files `cstartup.asm` and `rtx_conf.asm` must be configured.

The file `cstartup.asm` initializes all stack pointers, reserves code memory for the interrupt routines and sets the PWM0 output to zero (PWM0=0xFF). Table 1 contains a list of variables and their values used in the file `cstartup.asm`. For more details about the variables please refer to paragraph 6.10 CONFIGURATION FILES of the C51 Compiler User's Guide[3].

TABLE 1 Variables in cstartup.asm

Variable name	Value
IDATALEN	0x0100
XDATASTART	0x0000
XDATALEN	0x8000
PDATASTART	0x0000
PDATALEN	0x0000
IBPSTACK	0x0000
IBPSTACKTOP	0x0100
XBPSTACK	0x0000
XBPSTACKTOP	0x8000
PBSTACK	0x0000
PBSTACKTOP	0x8000
PPAGEENABLE	0x0000
PPAGE	0x0000

The RTX-51 operating system can be adapted to various members of the 8051 processor family and to application specific requirements by means of the file `rtx_conf.asm`. The following system values can be configured:

- Size of the standard and re-entrant task stack
- 8051 hardware timer to be used for the system clock
- Task switching with or without round-robin scheduling
- Type of the 8051 processor used

The variable `PROC_TYP` sets the processor type used. The processor types that can be used and the corresponding values of `PROC_TYP` are shown in table 2, other values of `PROC_TYP` are invalid.

TABLE 2 Processor types in rtx_conf.asm

PROC_TYP	Processor
1	8051, 8031, 8751, 80C31, 80C51, 87C51
2	80C521, 80C32
3	80515, 80C515, 80535, 80C535
4	80C517, 80C537
5	80C51FA/FB, 83C51FA/FB, 87C51FC
6	80C552, 83C552
7	80C592, 83C592, 87C592
8	80C152, 83C152
9	80C517A, 80C517A-5
10	80C652, 83C652
11	86C410, 86C610
12	80C550, 83C550, 87C550
13	80C51GB, 83C51GB, 87C51GB
14	88F51FC, 83F51FC
15	80512/80532
20	83CL580

Note: When another processor type (than 83CL580) is used please change the `reg580.inc` file accordingly.

Table 3 contains a list of system constants and their values as defined in `rtx_conf.asm`, for more details about the system constants please see chapter 9 CONFIGURATION of the RTX-51 User's Guide[4].

TABLE 3 System constants in rtx_conf.asm

System Constant name	Value	Meaning
?RTX_SYSTEM_TIMER	0	Use Timer 0 as system timer
?RTX_IE_INIT	0	All bits used
?RTX_IEN1_INIT	0	All bits used
?RTX_IEN2_INIT	0	All bits used
?RTX_INTSTKSIZE	64	Internal RAM
?RTX_EXTSTKSIZE	64	External RAM
?RTX_EXTRENTSIZE	50	External RAM (not used)
?RTX_TIMESHARING	0	Do not use round robin scheduling
?RTX_BANKSWITCHING	0	Code-Bank-Switching is disabled

2.7 Configurating the PSD312L

The PSD312L contains 64k bytes ROM, 2k bytes RAM and has 16 bidirectional I/O ports. Of these 16 bidirectional I/O ports 8 are used to follow A0 until A8 which are connected to the external 32k bytes RAM. The external RAM memory map of the system is shown in figure 1.

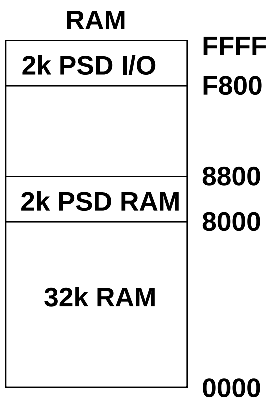


Fig.1 External RAM Memory map

To configure the PSD312L so that it can be connected to the 83CL580 the next items must be initialized using the PSD-Gold/PSD-Silver software:

- Mixed Address/Data Mode
- 8 bit Data Bus Size
- LOW reset polarity (always when using the L version)
- HIGH ALE polarity
- PSEN is used
- Use separate Data and Program Address spaces
- Port A to addressed I/O, all CMOS and PA0 corresponds to A0, PA1 to A1, PA2 to A2, PA3 to A3, PA4 to A4, PA5 to A5 PA6 to A6 and PA7 to A7
- Port B to I/O, all CMOS
- A19 is used for Chip Select Input (CSI)
- RAM memory address (RS0) is 0x8000
- PSD I/O address (CSP or CSIOPORT) is 0xF800

In Table 4 the PSD312L configuration bits and their values are shown.

TABLE 4 PSD312L Configuration bits

Configuration bit	Value
CDATA	0
CADDRDAT	1
CA19/CSI	0
CALE	0
COMB/SEP	1
CPAF2	0

TABLE 4 PSD312L Configuration bits

Configuration bit	Value
CADDHLT	0
CLOT	0
CRRWR	0
CEDS	0
CADLOG19	0
CPAF1	11111111
CPBF	11111111
CPCF	111
CPACOD	00000000
CPBCOD	00000000
CADLOG	000

3. Using the OM4754 software

The OM4754 software consists of a library file `(amps.lib)` and several source files. Table 5 gives an alphabetical list of files delivered as source and their contents.

TABLE 5 Delivered source files

File name	Contents
3wire.c	Three wire bus driver
3wire.h	Three wire bus driver definition file
aproc.c	SA5752/53 APROC driver
aproc.h	SA5752/53 APROC driver definition file
aud_tsk.c	Audio control Task main routine
audio.c	Audio Task
audio.h	Audio Task definition file
bindef.h	Binary definition file
car_m_k.h	Car mounting kit definition file
cstartup.asm	Initialization of RAM at start-up
disp_tsk.c	Display driver Task
dproc.c	UMA1000LT DPROC driver
dproc.h	UMA1000LT DPROC driver definition file
dprocint.c	UMA1000LT DPROC driver interrupt routine
eeprom.c	EEPROM driver
eeprom.h	EEPROM driver definition file
idle_tsk.c	Idle Task
iic.c	I^2C driver
iic.h	I^2C driver definition file
iic_int.c	I^2C driver interrupt routine
int_def.h	Interrupt definition file
io_utl.c	General I/O utilities file
io_utl.h	General I/O utilities definition file

TABLE 5 Delivered source files

File name	Contents
kb_tsk.c	Keyboard Task
lcd_drv.c	LCD driver
lcd_drv.h	LCD driver definition file
mail_box.h	Definition of messages sent via mailboxes
main.c	Main program to start RTX-51
mmi_disp.c	User Task display routines
mmi_disp.h	User Task display routines definition file
mmi_edit.c	User Task edit routines
mmi_edit.h	User Task edit routines definition file
mmi_func.c	User Task function mode routines
mmi_func.h	User Task function mode routines definition file
mmi_tsk.c	User Task
mmi_tsk.h	User Task definition file
on_off.h	ON/OFF definition file
p83cl580.c	P83CL580 initializations
p83cl580.h	P83CL580 initializations definition file
psd312.c	PSD312L driver
psd312.h	PSD312L driver definition file
random.c	Random generator
reg580.inc	P83CL580 special function register definition file
rtx_conf.asm	RTX-51 configuration
scnd_tsk.c	Second Task
sstm_tsk.c	System Task main routine
starttsk.c	Start-up Task
stateutl.h	Definition file for Standard routines for the System task
std_def.h	Standard definition file included in all C-files
synt.h	Synthesizer driver definition file
synt1015.c	UMA1015M Synthesizer driver
sysinit.c	Initialization routine for the System Task
sysvar.h	Definition file of the Global variables for the System Task
task_def.h	Task Identifier definition file
test.c	Test mode routines
test.h	Test mode routines definition file
timer.c	Timer driver
timer.h	Timer driver definition file
timerint.c	Timer driver interrupt routines
version.h	Version number definition file
vox_int.c	SA5752/53 APROC driver interrupt routine for VOX

Table 6 gives in alphabetical order, the contents of the library `amps.lib`.

TABLE 6 Files in library amps.lib

File name	Contents
access.obj	AMPS specification Chapter 2.6.3 System Access
convers.obj	AMPS specification Chapter 2.6.4 Mobile station Control on Voice Channel
idle.obj	AMPS specification Chapter 2.6.2 Idle
initial.obj	AMPS specification Chapter 2.6.1 Initialization
stateutl.obj	Standard routines for the System task
sysstat.obj	State machine for the System task
sysvar.obj	Global variables used by the System task
version.obj	Version number of the EAMPS software

3.1 Starting the system

In order to start the system a programmed PSD312L has to be inserted in the appropriate socket. The power can be switched on by pressing the ON/OFF key once. To switch the power off the ON/OFF key has to be pressed for about 1 second.

After the ON/OFF key has been pressed the system executes the `cstartup.asm` file which initializes all variables to zero and then calls the main program.

The main program (see `main.c`) initializes the PSD312L device and takes over the ON/OFF key by setting the PWR_ON bit. Then it initializes the micro controller specific registers (calling `p83cl580_INIT()` in `p83cl580.c`), the I^{2}C driver, the synthesizer driver and the random generator. When all these initializations are finished the RTX-51 operating system is started and the start-up task (see `starttsk.c)` is activated. When an error occurs during start-up of the RTX-51 operating system the system is switched off (calling `ms_turn_off()` in `main.c)`.

The start-up task sets the RTX-51 system clock, enables the I^{2}C interrupt, initializes the timer driver, initializes the DRPOC driver and reads the complete EEPROM contents to the RAM shadow area. Then the start-up task starts the idle, audio, display, second, keyboard, system and user tasks. When all tasks are started the start-up task will delete itself. If an error occurs when starting a task the system is switched off (calling `ms_turn_off()` in `main.c)`.

3.2 MMI description

For a complete description of the MMI please refer to the User Manual of the EAMPS demonstration and emulation unit[2].

4. Multitasking operating system

There are two fundamental problems for modern microprocessor applications:

- A task must be executed within a relative short time frame.
- Several tasks are time- and logic dependent from one another and should therefore execute simultaneously, but are executed on a single processor.

The first problem is also referred to the requirement for guaranteed response time, also designated as "real-time". The second problem designates the typical situation of multitasking operation. In this case, the individual tasks are organized as independent processes (also designated as tasks).

Therefore a multitasking operating system allows a group of tasks to cooperate in accomplishing an activity that can be parcelled into smaller concurrent activities. The multitasking operating system distributes the available micro processor time among the various tasks.

4.1 Tasks

During its existence a task goes through a series of discrete states. Various events can cause a task to change states. A process is said to be *running* if it currently has the CPU. A process is said to be *ready* if it could use a CPU if one were available. A process is said to be *blocked* if it is waiting for some event to happen (e.g. an I/O completion) before it can proceed.

On a single CPU system, only one task can be *running* at a time but several tasks may be *ready*, and several may be *blocked*. Therefore a ready list for ready tasks and a blocked list for blocked tasks is established.

When a task switches from one state to another a state transition has occurred. The states and transitions are displayed in figure 2.

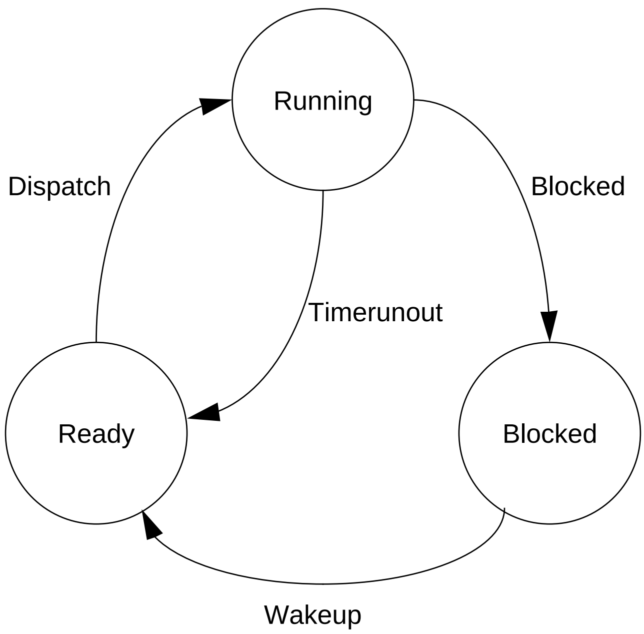


Fig.2 Task states transitions

As seen in the figure 2 there are four state transitions possible:

- Blocked; when a task waits for an event which is pending.
- Wakeup; when an event occurs for a task which was waiting for that event.
- Dispatch; when a task has a higher priority than the running task or when the running task blockes itself.
- Timerunout; when a task with a higher priority is put in the ready list.

When using the RTX-51 operating system the priority of a task can be 0, 1, 2 or 3. Value 0 corresponds to the lowest possible priority, value 3 corresponds to the highest possible priority. Priority 3 can only be used for fast task and is not used in this application.

4.2 Interrupt routines

The management and processing of hardware interrupts is one of the major jobs of the operating system. In this application standard C51 interrupt routines are used to interrupt the system. When an interrupt occurs, a jump is made to the corresponding interrupt routine directly and independent of the currently running task. The interrupt is processed outside of the operating system and therefore independent of the task scheduling rules.

However when an interrupt occurs a task would like to be informed that the interrupt has occurred. The task is informed via an event which is sent to that specific task. For the RTX-51 operating system the event is either a signal or a message.

Signals represent the simplest and fastest way of communication. When sending a signal no data is exchanged. The task number of the receiving task is used for identifying the signals for the individual operations.

By means of a mailbox concept, messages can be exchanged. Messages are exchanged in words (2 bytes). In this case, a message can represent the actual data to be transferred or the identification of a data buffer. In comparison to the signals, mailboxes are not assigned to a fixed task, but can be used freely by all tasks and interrupt routines.

4.3 Inter process communication

The policy of having an event driven operating system requires flexible means of inter process communication. The capability to move data from task to task is at the heart of the system functionality. Inter process communication is implemented via mailboxes.

Mailboxes are the interface between tasks which send messages to each other. Consequently, it is not necessary for a sender task to know anything about a receiver task's internal structure, or vice versa. This promotes a very clean and efficient mechanism for passing data.

The RTX-51 operating system provides a fixed number of eight mailboxes with a size of 2 bytes. If a task has to send more than 2 bytes, either a pointer has to be sent or the data should be stored in a global data array and a message should be sent to inform that the data has arrived. In the application there are 2 cases where the inter process communication is used to exchange data via a global data array, these cases are:

- send a received frame from DRPOC to the System task; A DPROC frame is 28 bits (4 bytes). A DPROC frame is stored in a global variable `received_frame` and a message is sent to the System task to inform the System task of the arrival of the DPROC frame.
- send the dialled number from the User task to the System task; A dialled number can be up to 32 digits. A dialled number is stored into a global variable `ddm_data` and a message is sent to the System task to inform the System task of the arrival of the dialled number.

The disadvantage of this method is that when a task is to slow in copying the data to his own local buffer the global data array could already be over written and thus corrupted.

5. The OM4754 software

The OM4754 software is designed to operate on a mobile that is build around the 83CL580 microcontroller, the SA5752/53 audio processor (APROC), the UMA1000LT data processor (DPROC), the ST25C02A EEPROM, the LP3800-A LCD and a RF system containing a.o. the UMA1015M synthesizer. The PSD312L programmable microcontroller peripheral is required to give the system the required amount of ROM. Figure 3 shows the mobile's hardware architecture. The relevant hardware parts with their 'connections' to the software are shown.

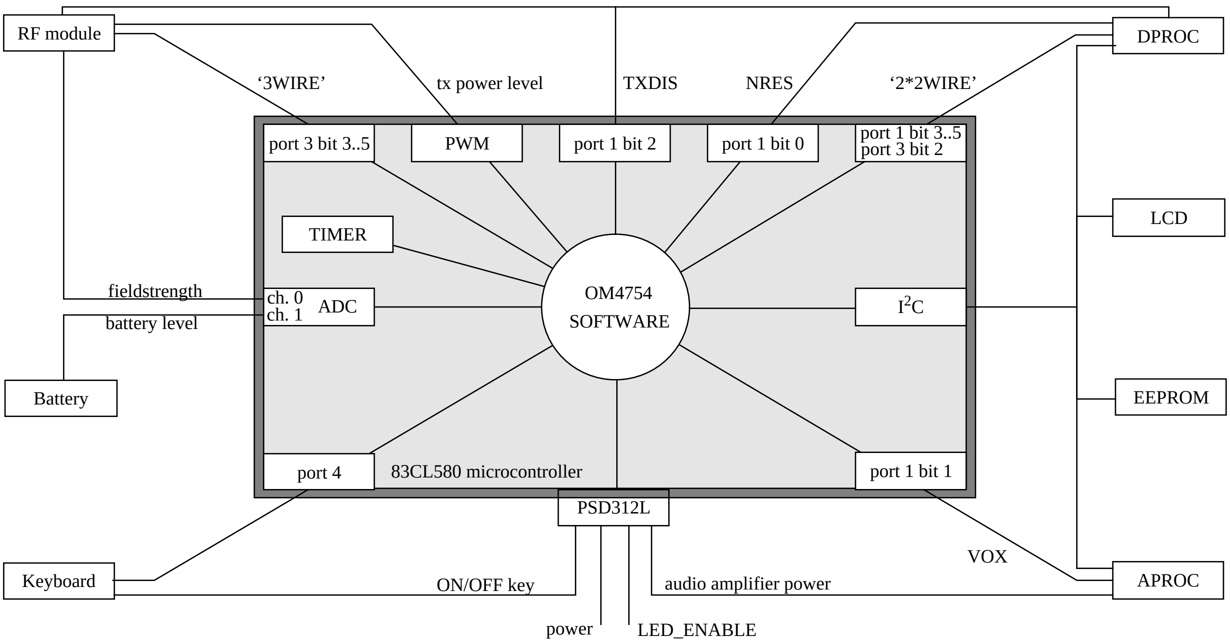


Fig.3 Hardware architecture

The OM4754 software uses 6 interrupt sources, in table 7 these interrupt sources are listed together with the interrupt number, vector address and file name. In table 7 the interrupt sources are listed in order of priority, the highest priority is the first interrupt listed, the lowest priority is the last interrupt listed.

TABLE 7 Interrupt sources

Interrupt source	Interrupt number	Interrupt vector	File name
I^{2}C port	5 (S1)	0x002B	iic_int.c
Timer 0 RTX-51 clock	1 (T0)	0x000B	
Timer 2 fast (1mS) timer	6 (T2)	0x0033	timerint.c
External 7 DPROC rx_line	12 (X7)	0x0063	dprocint.c
Timer 1 slow (20mS) timer	3 (T1)	0x001B	timerint.c
External 3 VOXout	8 (X3)	0x0043	vox_int.c

5.1 Software Tasks

The OM4754 software uses 8 different tasks. The Start-up task, is only used to start-up the system and deletes itself after it has started the other tasks. In table 8 the task names, their priorities, the mailbox names and if needed their special usage are listed.

TABLE 8 Task priorities

Task name	priority	Mailbox	Special Usage
Idle task	0		
Display task	1	MBX_DISPLAY	
Second task	1	MBX_SECOND	
Keyboard task	1	MBX_KEYBOARD	
Start-up task	2		
Audio task	2	MBX_AUDIO	
User task	2	MBX_USER IIC_SR_MBX	 EEPROM driver
System task	2	MBX_SYSTEM MBX_SYSTEM_TIMEOUT	 DPROC driver

All task definitions can be found in the file `task_def.h`. In the next sections the individual task are described in more details.

5.1.1 The Start-up task

The Start-up task is the first task that is called after the RTX-51 operating system has started and can be found in the file `starttsk.c`. The start-up task sets the RTX-51 system clock, enables the I^2C interrupt, initializes the timer driver, initializes the DRPOC driver and reads the complete EEPROM contents to the RAM shadow area. Then the Start-up task starts the Idle, Audio, Display, Second, Keyboard, System and User tasks. When all tasks are started the Start-up task will delete itself. If an error occurs when starting a task the system is switched off (by calling `ms_turn_off()` in the file `main.c`).

5.1.2 The System task

This task contains the state machine of the signalling software according the AMPS specification[1]. The System task can be found in the source files `sstm_tsk.c` and `sysinit.c` and the library file `amps.lib`.

The four main functions of a mobile in respect to signalling are registration, order response, page response and origination.

When the mobile is powered on, it searches for the strongest control channel, tunes to that channel and starts receiving and processing messages on that channel.

When a registration id message is received, the mobile checks to see whether a (autonomous) registration is required. This is required when the mobile roamed to another area since the last time it registered and when the registration id received in the registration message has been increased 'registration increment' times since the last registration. To register, the mobile accesses the reverse control channel and replies to the land station. It then waits for a registration confirmation message.

When the phone enters a service area where the registration bit is ‘1’, and when the registration bit received from the land station is changed from ‘0’ to ‘1’, the phone initiates a (non-autonomous) registration.

When an audit order is received, the mobile accesses the reverse control channel and replies to the land station.

When a page message is received, the mobile is being called. It accesses a reverse control channel and replies to the land station. The land station then supplies a voice channel number. The mobile changes to that voice channel and starts ringing. When the user accepts the call, the audio is switched through and the connection has been established.

When the user wants to start a call, the mobile accesses a reverse control channel. The number to be called is sent to the land station. The land station supplies a voice channel number. The mobile changes to that voice channel and the audio is switched through.

When the user wants to start an outgoing call while the mobile is accessing the reverse control channel for another reason (answering an incoming call, registration, order reply), the outgoing call gets priority and is made.

While a call is in progress, (the mobile is on a voice channel,) the SAT colour code (SCC) is checked. When it is not the expected SCC, the audio is muted. When the SCC remains wrong for more than 5 seconds, the call is aborted.

When a call is in progress it can be released by the user (mobile release) or from the land station (land release.)

When the mobile is on a voice channel, hand-off messages received are processed. The mobile switches to the new voice channel received in the hand-off message.

When a new power level is received from the land station, the mobile is set to transmit with the new power level.

Flash requests from the user during calls are executed.

The next items should be taken into account:

- The variables NXTREG_sp and SID_sp are stored in RAM, whenever the mobile is switched on these variables are initialized to 0, although section 2.3.4 of the AMPS specification[1] states that these variables should be stored at least for 48 hours after the mobile is switched off.
- The version number of the signalling software is always present in the character array `VS_array`; when an user program wants to obtain the version number the file `version.h` has to be included.
- The system task is not delivered as source but is present in the library file `amps.lib`.
- All system variables can be obtained when the file `sysvar.h` is included; The variable names in the AMPS specification[1] are the same as used in `sysvar.h`.
- The System task is set-up in the file `sstm_tsk.c`, this file is therefore delivered in source.
- The mobile is set-up in the file `sysinit.c`, this file is therefore delivered in source.

In `sysinit.c` the Station Class Mark of the mobile is set to binary 1110 which means that the mobile is a power class 3, discontinuous, 25MHz bandwith mobile. The variable `PA_switch_on_time` is read from EEPROM. This variable specifies the time the System task waits after the power amplifier is switched on before a message is sent (see AMPS specification[1] section 2.1.2.1). This time is only used when accessing the system (see also page 2-18 of the AMPS specification[1]). The variables PREFSYS_p, FIRSTCHC_p and ALTCHC_p in the file `sysinit.c` define whether the system type of the mobile is either A preferred, B preferred, A only or B only. In Table 9 the system type and the corresponding value of the variables PREFSYS_p, FIRSTCHC_p and ALTCHC_p are shown.

TABLE 9 System types

System type	PREFSYS_p	FIRSTCHC_p	ALTCHC_p
A Preferred	1	333	334
B Preferred	0	334	333
A Only	1	333	0
B Only	0	334	0

When the home only option is enabled, only the system corresponding to the least significant bit of the home system id is scanned, and service from other areas than the home area is rejected.

5.1.3 The User task

Although the display of data and the reading of the keyboard is done in the display and keyboard task respectively this task performs the dialog with the user. The User task can be found in the file `mmi_tsk.c`. The User task calls several routines which are located in the files `mmi_disp.c`, `mmi_edit.c`, `mmi_func.c` and `test.c` and their corresponding include files (like `mmi_disp.h`).

There are four points in the user task which are vital, these points are:

- Send DTMF tones
- Mute the Audio
- Start and stop the System task
- Send the dialled number from the User task to the System task

In order to send DTMF tones the event EV_DTMF has to be sent. The task where it should be sent to depends on the state in which the mobile is. If the mobile is in the conversation state the event EV_DTMF must be sent to the System task in order to power up the transmitter when the mobile is in discontinues transmission. The System task will send the EV_DTMF to the Audio task after the transmitter is powered up. In all other cases the event EV_DTMF must be sent directly to the Audio task.

When the user wants to mute the audio the event EV_SPEECH_PATH with the parameter value AUDIO_TX_MUTE for the transmitter or AUDIO_RX_MUTE for the receiver must be sent to the Audio task. To unmute the audio the event EV_SPEECH_PATH with the parameter value AUDIO_TX_UNMUTE for the transmitter or AUDIO_RX_UNMUTE for the receiver must be sent. The parameter values AUDIO_MUTE and AUDIO_UNMUTE are reserved by the System task to mute the audio and should therefore not be used by any other task.

To start or stop the System task the events EV_HALT_SYSTEM and EV_TURN_OFF_REQUEST can be used. The EV_HALT_SYSTEM halts the system task until a signal is received, the EV_TURN_OFF_REQUEST stops the System task completely the only way out is to reboot the system. When the User task wants to stop the System task the mobile must not be in conversation. If the mobile is in conversation first an event EV_END_PRESSED must be sent to the system task, the User task must than wait for the event EV_CONVERSATION with the parameter value CONV_END or NO_CONNECT before it can stop the System task.

When the user presses the SEND key an origination or a flash request is made. The User task copies the dialled number in the `ddm_data` buffer and sends a message EV_SND_PRESSED to the System task. The dialled number in the `ddm_data` buffer is presented in ASCII characters, the System task will convert the ASCII characters to the required digit code as described in table 2.7.1-2 of the AMPS specification[1].

A complete list of all events and their parameter values are given in chapter 5.2.

For a complete description of the MMI please refer to the User Manual of the EAMPS demonstration and emulation unit[2].

5.1.4 The Audio task

The Audio task controls the generation of sounds, audio volume, setting of the audio path and the audio amplifier. Sounds that are generated include alarms and dtmf sounds. Generation of sounds and controlling both the audio path and volume is done by setting registers in the audio processor. The Audio task can be found in the file `aud_tsk.c`, the main audio routine can be found in `audio.c`.

The mobile can operate hand-held or hands-free and can be connected to a car kit. For each of these 'modes' the volume has individual settings. A full car kit implementation is not given.

Audio is received from and transmitted to the land station using a pair of voice channels; a forward voice channel and a reverse voice channel. The audio path is the path between user and voice channel. The audio path can be muted independently in the transmit direction (tx-path) and in the receive direction (rx-path).

When the mobile is operated hand-held or is connected to the car kit, the audio is output to the earpiece. When the mobile is operated hands-free the audio is output to the loudspeaker. When a car kit is not connected, the audio amplifier is turned on.

The volume can be changed between its minimum (0) and its maximum (15) value and is stored for each one of the four possible audio modes: hand-held, hands-free, car-kit and internal. Whenever the audio mode is changed, the volume level is restored to the volume level of the previous audio mode. When the mobile is turned on the volume level for all four modes is set to their defaults.

Alarms and dtmf tones are generated using the dtmf generator in the audio processor. When an alarm or dtmf tone is generated, the audio is output to the loudspeaker when the car kit is not connected. When the car-kit is connected, the audio is output to the earpiece. When the alarm or dtmf tone finishes, the audio is output again to where it was output before the alarm. During generation of alarms the tx-path is muted.

All alarms have a priority. When an alarm is started, first a check is done to see whether an alarm with a higher priority is already active. When an alarm with a higher priority is active the new alarm is not started. In Table 10 the different alarms and their duration are given. The alarms are listed in order of priority, the highest priority is the first alarm listed, the lowest priority is the last alarm listed.

TABLE 10 Audio alarms

Alarm type	Toggle time	Duration	Tone	Volume
malfunction	300 mS	3 Seconds	DTMF_MALFUNCTION	4/8 (See note 1)
ringing	50 mS	65 Seconds	DTMF_HIGH/LOW_TONE/SPACE	Ringing volume
low_voltage	50 mS	1 Second	DTMF_LOW_VOLTAGE	8
wake_up	100 mS	1 Second	DTMF_WAKE_UP	8
key_beep/DTMF		100 mS	Depending on key pressed	Key volume
call_setup		200 mS	DTMF_WAKE_UP	4
service_area		200 mS	DTMF_WAKE_UP	4

Note 1: Depending on the setting of the `malfunction_loudness`. The `malfunction_loudness` can be changed using the event EV_MALFUNCTION_LOUDNESS. The default value of `malfunction_loudness` is HIGH which corresponds to volume 8.

For all toggle time's listed the alarm is switched on/off for the specified toggle time period. For the ringing alarm the tone is switched between the three listed tone's for every toggle time period.

5.1.5 The Keyboard task

The Keyboard task can be found in the file `kb_tsk.c`. The Keyboard task scans every 40 milli seconds the keyboard for pressed keys. Keys found to be pressed are sent to the User task. The CLEAR key is repeated when pressed longer than 400 milli seconds, the ON/OFF key is repeated as soon as it stays pressed. The keyboard has a higher priority than the ON/OFF key, which means that when a key is pressed together with the ON/OFF key the key pressed is processed. As soon as a repeated key is released a key release message is sent to the User task. The keyboard is connected to port 4 of the microcontroller. The ON/OFF key is connected to the PSD312L.

5.1.6 The Display task

The Display task displays the information received from System, Second and User task on the LCD display. It computes the events received to function calls of the LCD driver. The Display task can be found in the file `disp_tsk.c` and the LCD driver in the file `lcd_drv.c`. The LCD driver is not explained here but in section 5.4.

5.1.7 The Second task

The (one) Second task preforms tasks that can be considered 'continuous', but do not have a high priority. The Second task can be found in the file `scnd_tsk.c`. Every second the Second task measures the fieldstrength of the received signal for display purposes, and monitors the battery level. When the battery level drops below the warning or turn off threshold for at least 3 consecutive measurements, a message is sent to the User task.

5.1.8 The Idle task

The Idle task is the task that is active when all other tasks are blocked. The Idle task is the task which has the lowest priority possible. The Idle task can be found in the file `idle_tsk.c`. When this task is activated the micro controller is switched to IDLE mode. The micro controller can only wake up from the IDLE mode via an interrupt. The IDLE mode is only entered if the file `idle_tsk.c` is compiled with the compiler switch `PRODUCTION_PHONE`. The reason is that when using an emulator the bond out chip P85CL001 will only work till 3.7 Volts at 9.6 MHz, the 83CL580 however is supplied with 3.5 Volts. Therefore it is important that the IDLE mode is not used when using an emulator otherwise the system performance will degrade to an undesired level!

5.2 Message types and formats

Events are sent using the os_send_message or isr_send_message system call. The arguments to these calls are described in the RTX-51 documentation. The 2 byte message contains the event (first byte) and optionally a parameter (second byte). The events and optional parameters can be found in the file `mail_box.h` and are described in the next paragraphs.

5.2.1 Events sent to the User task.

EV_KEYPRESS

The EV_KEYPRESS is sent when the user presses a key on the keyboard.

From	: Keyboard task	
To	: User task	
Parameter	: key	
Description	: this byte contains the key that the user has pressed on the keyboard.	
Values	: KEY_EMPTY = ‘ ‘	KEY_ZERO = ‘0’
	KEY_ON_OFF = ‘O’	KEY_ONE = ‘1’
	KEY_CLEAR = ‘C’	KEY_TWO = ‘2’
	KEY_MUTE = ‘M’	KEY_THREE = ‘3’
	KEY_STORE = ‘S’	KEY_FOUR = ‘4’
	KEY_RECALL = ‘R’	KEY_FIVE = ‘5’
	KEY_SEND = ‘W’	KEY_SIX = ‘6’
	KEY_END = ‘E’	KEY_SEVEN = ‘7’
	KEY_FUNCTION = ‘F’	KEY_EIGHT = ‘8’
	KEY_UP = ‘+’	KEY_NINE = ‘9’
	KEY_DOWN = ‘-’	KEY_STAR = ‘*’
	KEY_OK = ‘K’	KEY_HASH = ‘#’
Comments	:	
See also	: EV_KEYREPEAT, EV_KEYRELEASE	

EV_KEYREPEAT

The EV_KEYREPEAT is sent when a key is pressed longer than its initial delay. An EV_KEYREPEAT is generated repetitive until the key is released.

From	: Keyboard task
To	: User task

Parameter : key
Description : This byte contains the ASCII code for the key that has been repeated.
Values : see values for EV_KEYPRESS
Comments : Only the KEY_CLEAR and the KEY_ON_OFF are repeated. The initial delay for KEY_CLEAR is 400 msec, for KEY_ON_OF there is no initial delay. The repeat rate is 25 per second.
See also : EV_KEYPRESS, EV_KEYRELEASE

EV_KEYRELEASE

The EV_KEYRELEASE is sent when the user releases a key that has been pressed on the keyboard.

From : Keyboard task
To : User task
Parameter : key
Description : This byte contains the ASCII code for the key that has been released on the keyboard.
Values : see values for EV_KEYPRESS
Comments : this event is only generated when an EV_KEYREPEAT has been sent for the key released.
See also : EV_KEYPRESS, EV_KEYREPEAT

EV_LOW_VOLTAGE

The EV_LOW_VOLTAGE is sent when the battery voltage drops below the warning threshold for 3 seconds and is repeated every 3 seconds as long as it stays below that threshold.

From : Second task
To : User task
Parameter : battery_alarm_level
Description : The threshold which was reached.
Values : enum { LOW_VOLTAGE_WARNING,
LOW_VOLTAGE_TURN_OFF } ;
Comments :
See also :

EV_CONVERSATION

From : System task
To : User task
Parameter : conversation_status
Description : The conversation status the mobile has entered.
Values : enum { RINGING_CALL,
SILENT_CALL,
CONV_START,
CONV_END,
MOBILE_RELEASE,
LAND_RELEASE,
SYSTEM_BUSY,
NO_CONNECT,
INTERCEPT,
IGNORED,
FLASH_SEND };
Comments :
See also :

5.2.2 Events sent to the System task.

EV_FRAME_RECEIVED

The EV_FRAME_RECEIVED is sent whenever a frame is received from the DPROC.

From	: DPROC interrupt routine	
To	: System task	
Parameter	: frame_type	
Description	: The type of frame received.	
Values	: enum {	Abb_add_word = 1, Ext_add_word_1, Ext_add_word_2_order, Ext_add_word_2_order_wrong_min, Ext_add_word_2_chan, Ext_add_word_2_chan_wrong_min, Sys_par_over_mess_1, Sys_par_over_mess_2, Global_action_mess, Reg_id_mess, Control_filler_mess };
Comments	: The frame received is put at the global memory location designated during initialisation of the DPROC driver.	
See also	:	

EV_VOICE_DETECT

The EV_VOICE_DETECT is sent whenever a change in the presence of voice has been detected.

From	: VOX interrupt routine	
To	: System task	
Parameter	: detected	
Description	: The change in the presence of voice.	
Values	: enum {	VOICE_DETECTED, SILENCE_DETECTED };
Comments	:	
See also	:	

EV_SND_PRESSED

From	: User task
To	: System task
Parameter	: none
Description	:
Comments	: When doing an origination or a flash, the dialled number must be stored in the dialled number buffer, ddm_data.
See also	:

EV_END_PRESSED

From	: User task
To	: System task
Parameter	: none
Description	:
Comments	:
See also	:

EV_ALLOW_DTX

From : User task
To : System task
Parameter : dtx_setting
Description : The setting of DTX which was changed.
Values : enum { ALLOW_DTX,
INHIBIT_DTX };
Comments :
See also :

EV_INIT_SYSTEM

From : User task
To : System task
Parameter : none
Description :
Comments : Used to restart the signalling part. Can be used when changing country.
See also :

EV_HALT_SYSTEM

From :
To : System task
Parameter : none
Description :
Comments : The system task will be resumed when a signal is sent to it.
See also :
Example : To halt the system task : `RTX_send_message(MBX_SYSTEM, EV_HALT_SYSTEM, 0, 0);`
To resume the system task : `RTX_send_signal(SYSTEM_TASK );`

EV_TURN_OFF_REQUEST

From : User task
To : System task
Parameter : none
Description :
Comments : Turns the system task off. The system task only gets active again after a ‘reboot’ of the system.
See also :

5.2.3 Events sent to the Audio task

EV_VOLUME

From : User task
To : Audio task
Parameter : direction
Description : The direction in which the volume has to be changed in.
Values : enum { VOLUME_UP,
VOLUME_DOWN };
Comments :
See also :

EV_VOLUME_ABS

From : User task
To : Audio task

Parameter : volume level
Description : Defines the absolute volume level.
Values : 0x00...0x0F
Comments :
See also :

EV_AUDIO_POWER

From : System task
To : Audio task
Parameter : audio_power
Description : Switches the TDA7050 Audio Amplifier ON/OFF.
Values : enum { ON,
OFF } ON_TYPE;
Comments :
See also :

EV_SPEECH_PATH

From : System task, User task
To : Audio task
Parameter : speech path control
Description : Defines the audio speech path.
Values : enum { AUDIO_MIC_MUTE,
AUDIO_HANDHELD,
AUDIO_HANDSFREE,
AUDIO_MUTE,
AUDIO_UNMUTE,
AUDIO_RX_MUTE,
AUDIO_RX_UNMUTE,
AUDIO_TX_MUTE,
AUDIO_TX_UNMUTE };
Comments :
See also :

EV_RINGING

Generates a standard length ringing tone.

From : System task
To : Audio task
Parameter : none
Description :
Comments : The standard length for a ringing tone is 65 seconds.
See also :

EV_TIMELENGTH_RINGING

Generates a ringing tone of the specified length.

From : System task
To : Audio task
Parameter : time length
Description : The time length of the tone to be generated.
Values : 0...254, DEFAULT_PARM
Comments : When time length is in the range of 0...254, a ringing tone is generated for a time of (time length * 20ms)
Otherwise a ringing tone of 65 seconds is generated.
See also :

EV_MALFUNCTION

Generates a standard length malfunction tone.

From	: User task
To	: Audio task
Parameter	: none
Description	:
Comments	: The standard time length for a malfunction tone is 3 seconds.
See also	:

EV_TIMELENGTH_MALFUNCTION

Generates a malfunction tone of the specified length.

From	: User task
To	: Audio task
Parameter	: time length
Description	: The time length of the tone to be generated.
Values	: 0...254, DEFAULT_PARM
Comments	: When time length is in the range of 0...254, a malfunction tone is generated for a time of (time length * 20ms). Otherwise a malfunction tone is generated for 3 seconds.
See also	:

EV_DTMF

Generates a standard length dtmf tone.

From	: User task		
To	: Audio task, System task		
Parameter	: tone		
Description	: The dtmf tone to generate.		
Values	: KEY_ONE = ‘1’	: 1209 Hz	697 Hz
	KEY_TWO = ‘2’	: 1336 Hz	697 Hz
	KEY_THREE = ‘3’	: 1477 Hz	697 Hz
	KEY_FOUR = ‘4’	: 1209 Hz	770 Hz
	KEY_FIVE = ‘5’	: 1336 Hz	770 Hz
	KEY_SIX = ‘6’	: 1477 Hz	770 Hz
	KEY_SEVEN = ‘7’	: 1209 Hz	852 Hz
	KEY_EIGHT = ‘8’	: 1336 Hz	852 Hz
	KEY_NINE = ‘9’	: 1477 Hz	852 Hz
	KEY_ZERO = ‘0’	: 1336 Hz	941 Hz
	KEY_STAR = ‘*’	: 1209 Hz	941 Hz
	KEY_HASH = ‘#’	: 1477 Hz	941 Hz
	DTMF_MALFUNCTION = ‘m’	: 2000 Hz	-
	DTMF_LOW_VOLTAGE = ‘v’	: 2273 Hz	-
	DTMF_HIGH_TONE = ‘h’	: 1010 Hz	-
	DTMF_LOW_TONE = ‘l’	: 800 Hz	-
	DTMF_SPACE = ‘c’	: -	-
	DTMF_WAKE_UP = ‘w’	: 1000 Hz	-
	DTMF_STOP = ‘s’	: -	-
	‘A’	: 1633 Hz	697 Hz
	‘B’	: 1633 Hz	852 Hz
	‘C’	: 1633 Hz	852 Hz
	‘D’	: 1633 Hz	941 Hz
	“anything else”	: 1209 Hz	-
Comments	: The dtmf tones are generated for 96 ms. When the phone is in conversation and a DTMF tone has to be sent to the land station, the EV_DTMF has to be sent to the System task. In all other cases, the EV_DTMF can be sent to the Audio task.		

See also :

EV_TIMELENGTH_DTMF

Generates a standard length dtmf tone.

From : User task
To : Audio task
Parameter : tone
Description : The tone to be generated.
Values : See values EV_DTMF
Comments : The dtmf tones are generated for 100 ms.
See also :

EV_TIMELENGTH_KEY_BEEP

Generates a standard length dtmf tone.

From : User task
To : Audio task
Parameter : tone
Description : The dtmf tone to generate.
Values : See values EV_DTMF
Comments : Identical to EV_TIMELENGTH_DTMF
See also : EV_TIMELENGTH_DTMF

EV_TIMELENGTH_CALL_SETUP_TONE

Generates a call setup tone of the specified length.

From : System task, Audio task
To : Audio task
Parameter : time length
Description : The time length of the tone to be generated.
Values : 0...254, DEFAULT_PARM
Comments : When time length is in the range of 0...254, a call setup tone is generated for a time of (time length * 20ms). Otherwise a call setup tone is generated for 200 ms.
See also :

EV_TIMELENGTH_SERVICE_AREA_ALERT

Generates a service area alert tone of the specified length.

From :
To : Audio task
Parameter : time length
Description : The time length of the tone to be generated.
Values : 0...254, DEFAULT_PARM
Comments : When time length is in the range of 0...254, a service area alert tone is generated for a time of (time length * 20ms). Otherwise a service area alert tone is generated for 200 ms.
See also :

EV_TIMELENGTH_WAKE_UP_ALARM

Generates a wake-up alarm of the specified length.

From : System task
To : Audio task
Parameter : time length

Description : The time length of the tone to be generated.
Values : 0...254, DEFAULT_PARM
Comments : When time length is in the range of 0...254, a wake-up alarm is generated for a time of (time length * 20ms). Otherwise a wake-up alarm is generated for 1 second.
See also :

EV_TIMELENGTH_LOW_VOLTAGE

Generates a low voltage tone of the specified length

From : User task
To : Audio task
Parameter : time length
Description : The time length of the tone to be generated.
Values : 0...254, DEFAULT_PARM
Comments : When time length is in the range of 0...254, a low voltage alarm is generated for a time of (time length * 20ms). Otherwise a low voltage alarm is generated for 200 ms.
See also :

EV_STOP_AUDIO_ALARM

Stops generating the specified alarm.

From : System task, User task
To : Audio task
Parameter : alarm_type
Description : The alarm type to stop generating.
Values : enum { AUDIO_STOP_RINGING,
AUDIO_STOP_MALFUNCTION,
AUDIO_STOP_KEYBEEP,
AUDIO_STOP_SERVICE_AREA_ALERT,
AUDIO_STOP_CALL_SETUP_TONE,
AUDIO_STOP_WAKE_UP_ALARM,
AUDIO_STOP_LOW_VOLTAGE,
AUDIO_STOP_DTMF,
AUDIO_STOP_ALL };
Comments :
See also :

EV_VOX_ON

Turns on detection of voice presence.

From : System task
To : Audio task
Parameter : none
Description :
Comments :
See also : EV_VOX_OFF, EV_VOICE_DETECT

EV_VOX_OFF

Turns off detection of voice presence.

From : System task
To : Audio task
Parameter : none
Description :
Comments :

See also : EV_VOX_ON, EV_VOICE_DETECT

EV_RINGING_LOUDNESS

Sets the ringing volume.

From : System task
To : Audio task
Parameter : loudness
Description : The ringing loudness is set LOW/HIGH.
Values : enum { LOW,
HIGH };
Comments :
See also :

EV_MALFUNCTION_LOUDNESS

Sets the malfunction loudness.

From :
To : Audio task
Parameter : loudness
Description : The malfunction loudness is set LOW/HIGH.
Values : enum { LOW,
HIGH };
Comments :
See also :

EV_EXTERNAL_EQUIPMENT_CHANGED

From :
To : Audio task
Parameter : external_equipment
Description : The external equipment the mobile is now connected to.
Values : enum { AUDIO_CHANGE_TO_HANDHELD,
AUDIO_CHANGE_TO_CARKIT,
AUDIO_CHANGE_TO_LINEINTERFACE,
AUDIO_CHANGE_TO_EXT_ANTENNA };
Comments : This event is not implemented.
See also :

EV_POWER_OFF

Powers down the task.

From :
To : Audio task
Parameter : none
Description :
Comments : The system is about to power down, finish any jobs and save information.
See also :

5.2.4 Events sent to the Display task

EV_UPDATE_DISPLAY

Updates the alpha-numeric part of the display.

From : User task
To : Display task
Parameter :
Description :
Comments :
See also :

EV_UPDATE_SYMBOL

From : System task, User task
To : Display task
Parameter : symbol
Description : The symbol together with a modifier attribute.
Values :

```
/* The symbols */
enum {      MENU_SYMBOL,
            BOOK_SYMBOL,
            ROAM_SYMBOL,
            NO_SYMBOL,
            SERV_SYMBOL,
            IN_USE_SYMBOL,
            LOCK_SYMBOL,
            A_SYMBOL,
            B_SYMBOL,
            VOX_SYMBOL,
            HANDS_FREE_SYMBOL,
            FUNC_SYMBOL,
            ALPHA_SYMBOL,
            MUTE_SYMBOL,
            SIG_SYMBOL };
/* The modifier attributes */
enum {      OFF_MODE = 0x00,
            FLASH_MODE = 0x40,
            FLASH_REVERSE_MODE= 0x80,
            ON_MODE = 0xC0 };
```

Comments :
Example : `RTX_send_message(MBX_DISPLAY, EV_UPDATE_SYMBOL, ROAM_SYMBOL | ON_MODE, 0);`
See also :

EV_UPDATE_FS_VALUE

Updates the field strength value.

From : Second task
To : Display task
Parameter : field strength
Description : The (relative) field strength of the received signal.
Values : 0...255
Comments :
See also :

EV_UPDATE_BATT_VALUE

From : Second task
To : Display task
Parameter : battery_value
Description : The (relative) battery value.
Values : 0...255
Comments : A value of 0 corresponds with 0 Volt and a value of 255 with 14.1 Volt. This event is not implemented.
See also :

5.2.5 Events sent to the Second task

EV_DISPLAY_RSSI

Disables or enable measurement of the RSSI.

From : System task
To : Second task
Parameter : rssi_enable
Description : Enable/Disable the RSSI update.
Values : typedef enum { DISABLE = 0,
ENABLE } ENABLE_TYPE;
Comments :
See also :

5.2.6 Events sent by drivers

EV_TIMEOUT

Expiration of a timer.

From : FAST TIMER and SLOW TIMER interrupt routines
To : any mailbox
Parameter : timer_handle
Description : The handle of the timer expired.
Values : 0...255
Comments : The mailbox the message is sent to was specified when the timer was started.
See also :

EV_IIC_TX_COMPLETE

The I^2C interrupt routine completed transmission of data on the I^2C bus.

From : I^2C interrupt routine
To : any mailbox
Parameter :
Description :
Comments :
See also :

EV_IIC_RX_COMPLETE

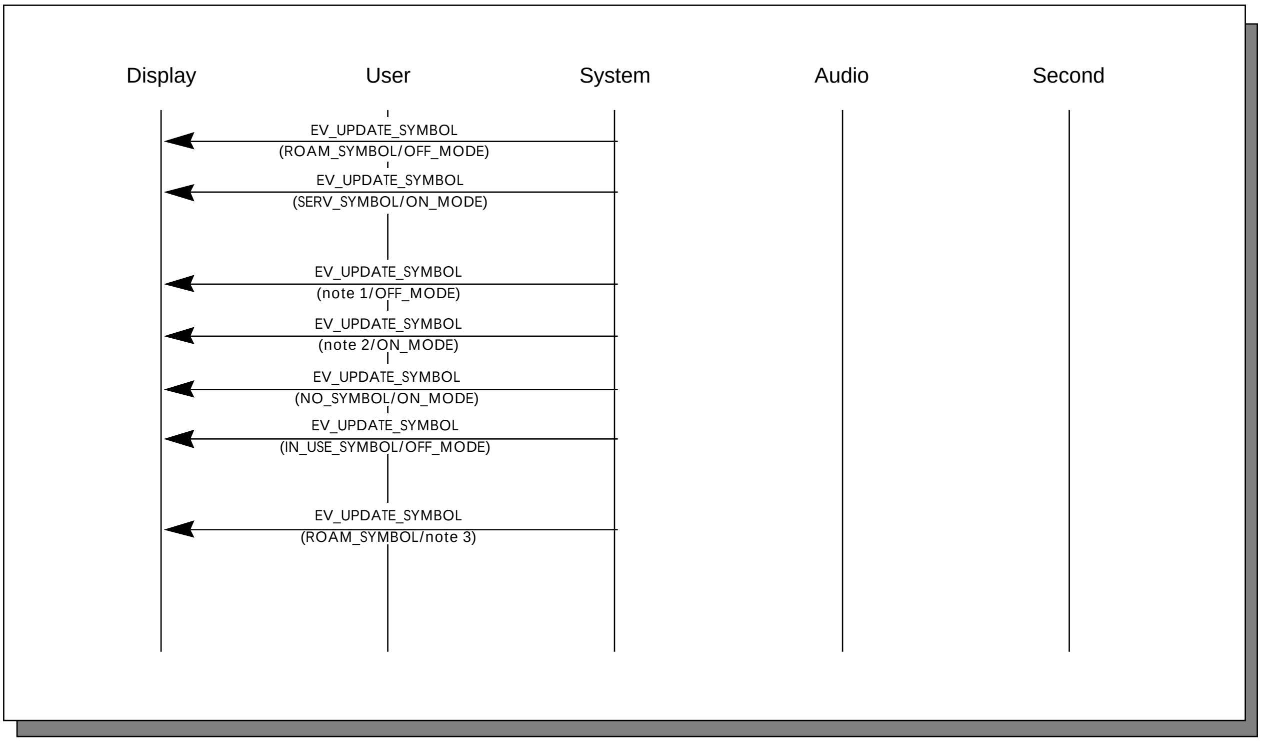
The I^2C interrupt routine received data on the I^2C bus.

From : I^2C interrupt routine
To : any mailbox
Parameter :
Description :
Comments :
See also :

5.3 Message scenarios

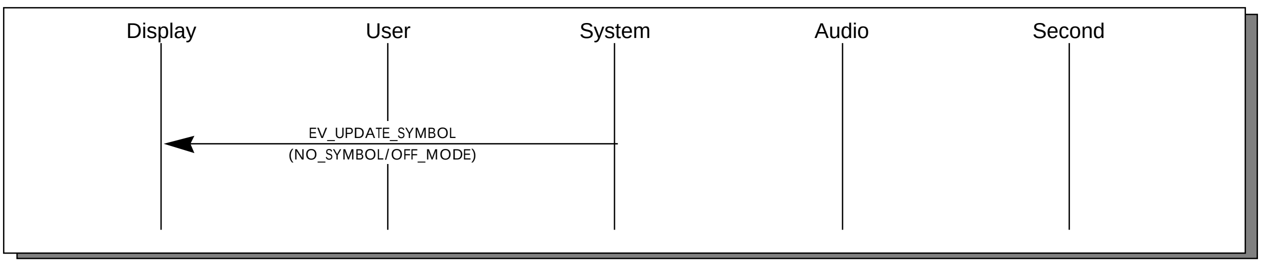
The following scenarios show the interactions between tasks in the OM4754 Software.

5.3.1 Initialisation



(1) When scanning system A symbol is B_SYMBOL, else symbol is A_SYMBOL.
(2) When scanning system A symbol is A_SYMBOL, else symbol is B_SYMBOL.
(3) When roaming mode is ON_MODE, else mode is OFF_MODE.

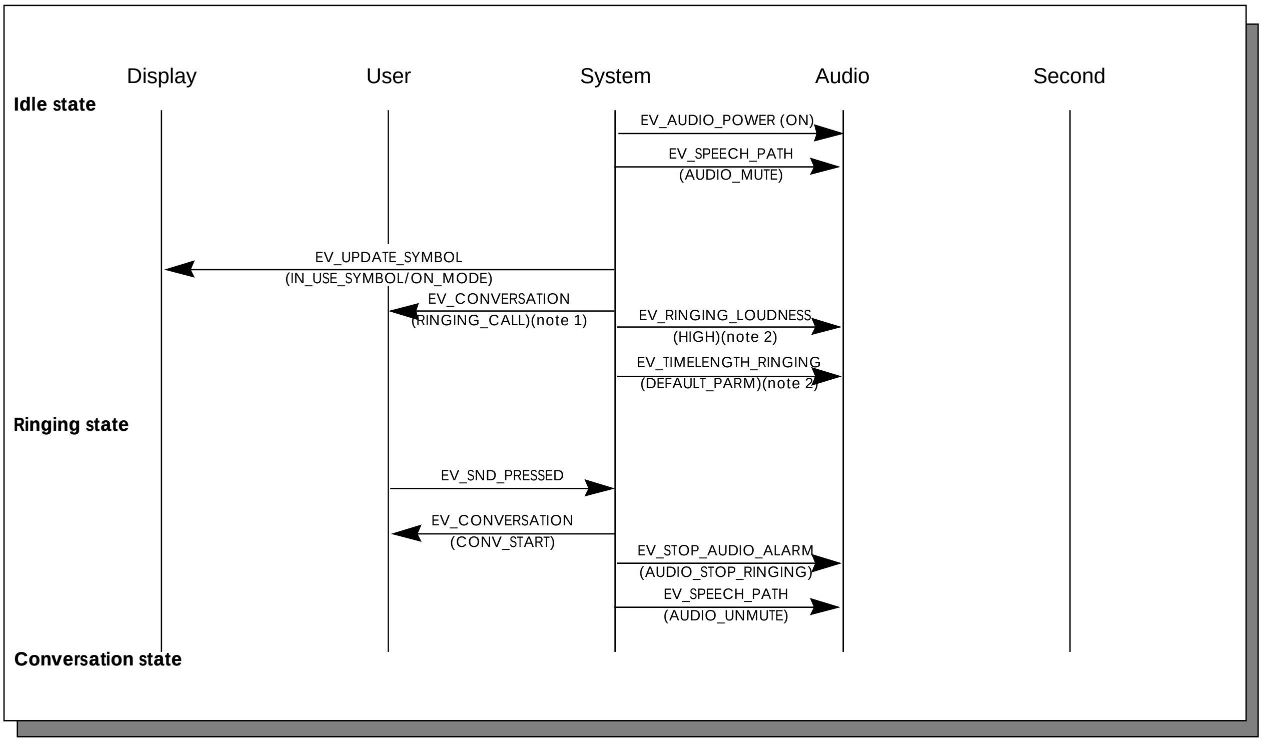
5.3.2 Finding Service



5.3.3 Paging

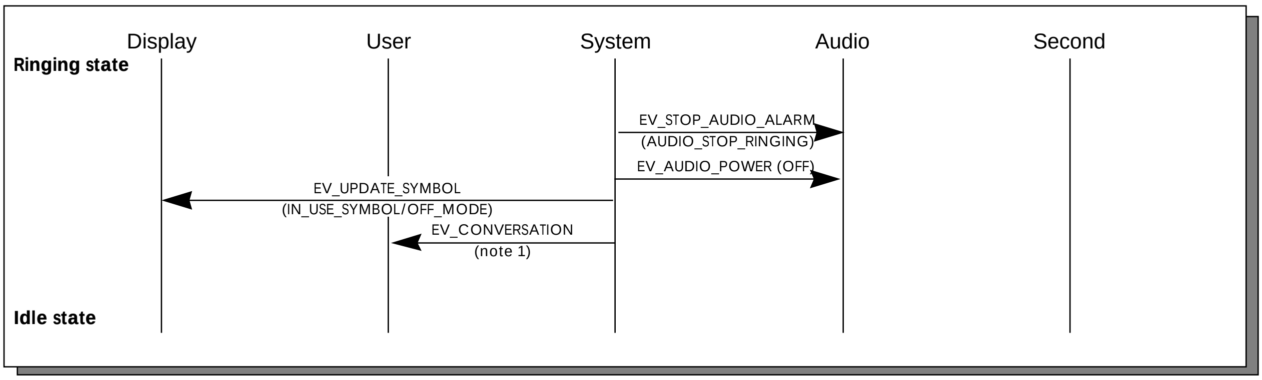
Successful Paging

Ringing Call / Silent Call



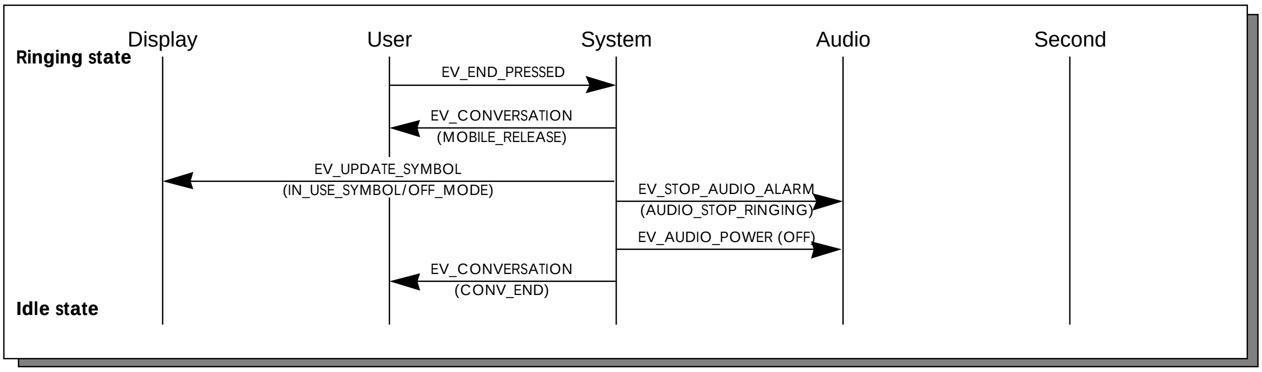
(1) When Silent Call this message is EV_CONVERSATION/SILENT_CALL.
(2) When Silent Call this message is not sent.

Paging Failed on Voice Channel



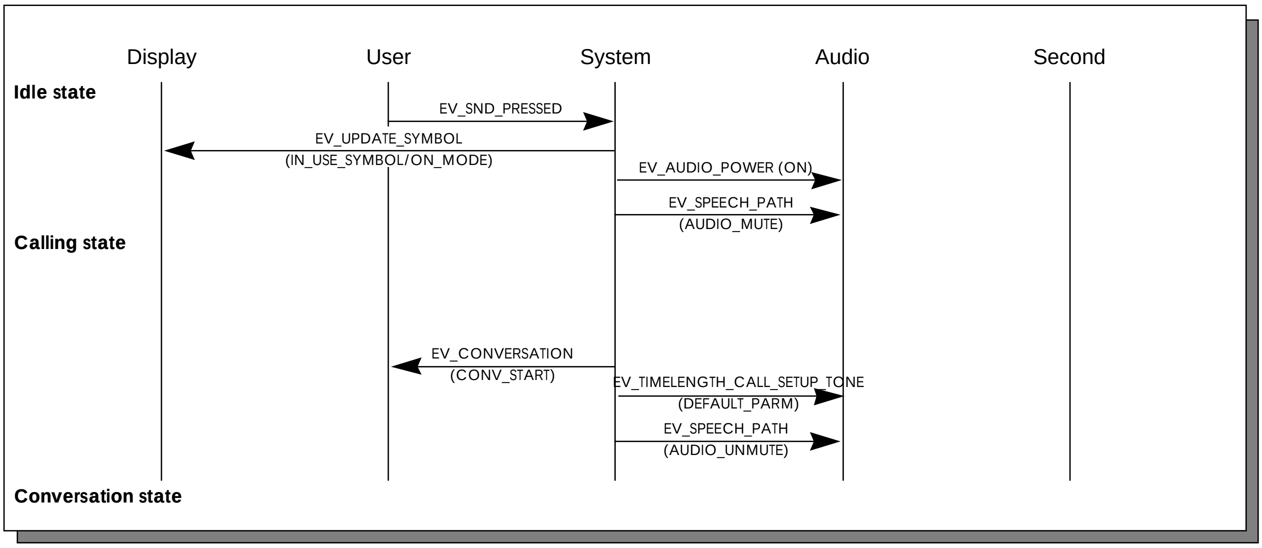
(1) The status can be SYSTEM_BUSY or NO_CONNECT.

User Declined Paging



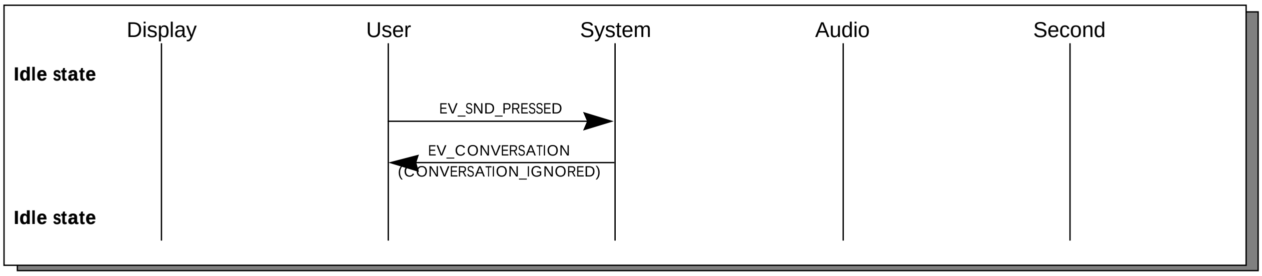
5.3.4 Origination

Successful Origination

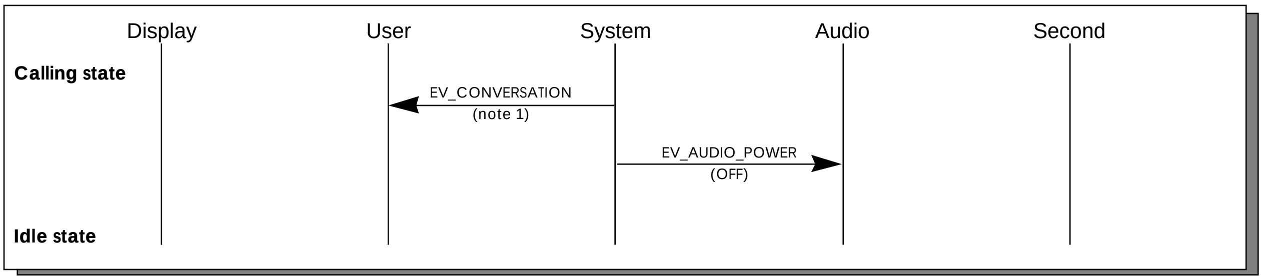


Failed Origination

Ongoing signalling

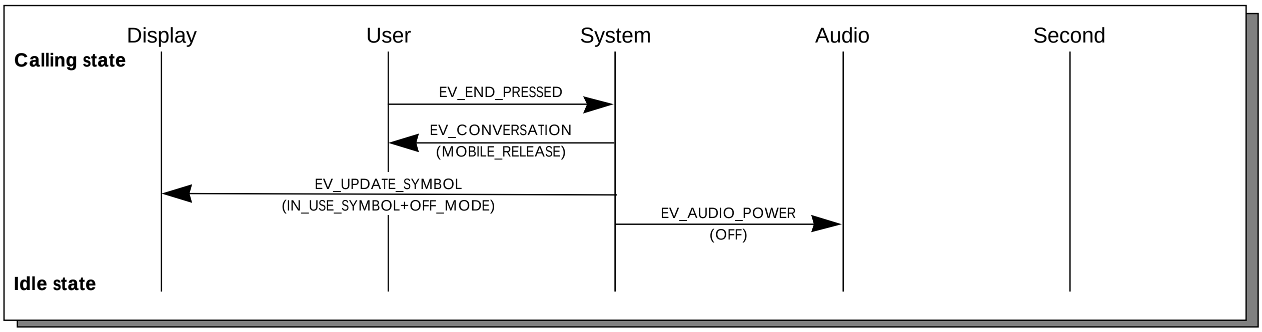


Connection Failure or System Busy



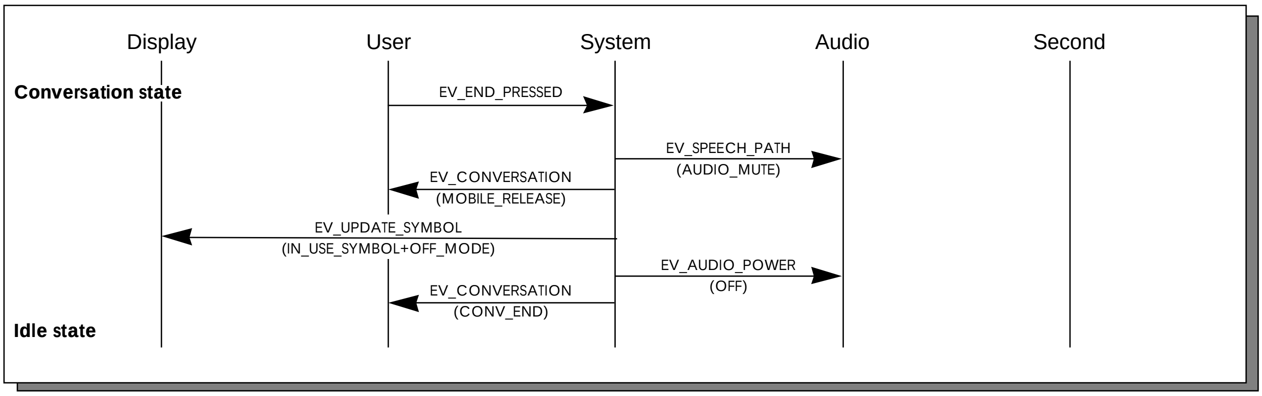
(1) The status can be SYSTEM_BUSY, NO_CONNECT or INTERCEPT.

User Declined Origination

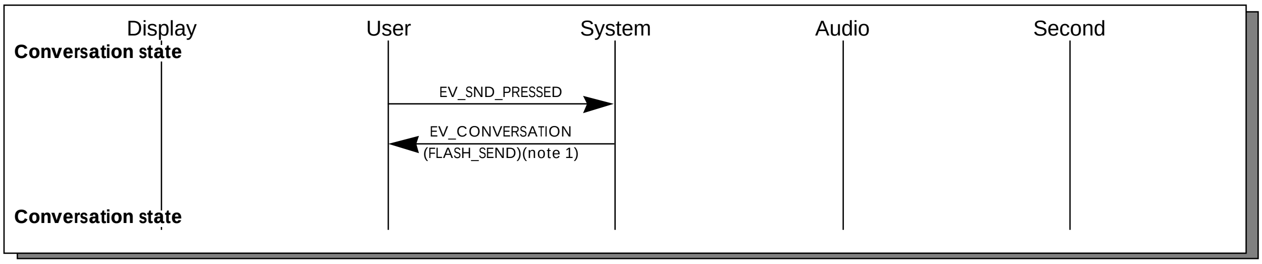


5.3.5 Conversation

Mobile Release



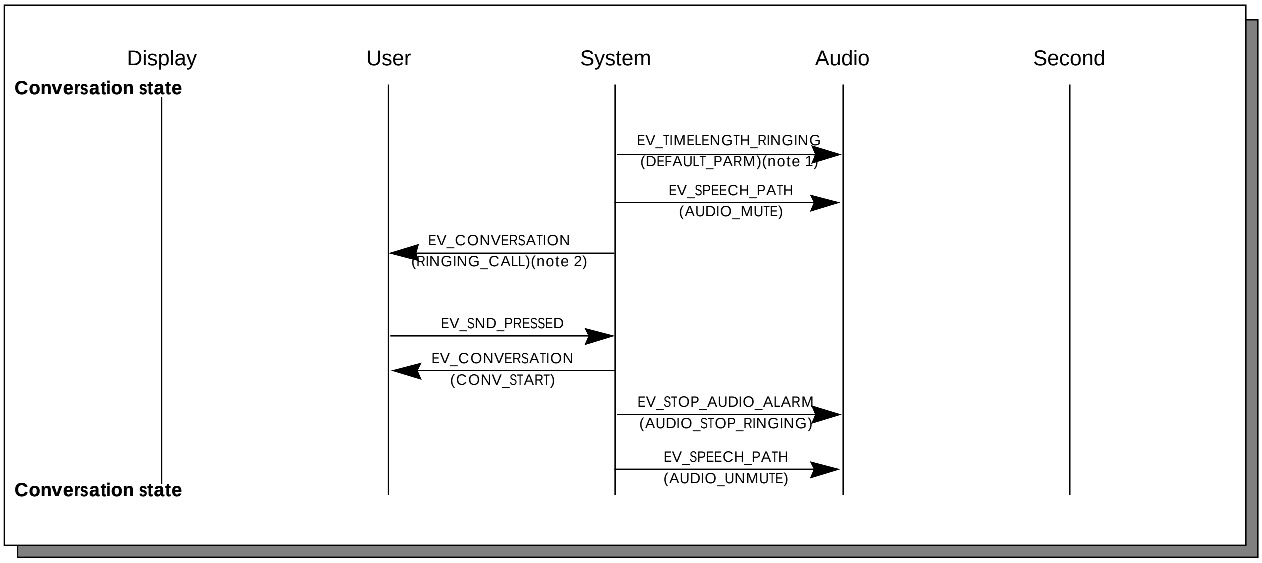
Successful Flash Request



(1) Message is sent when the number in the ddm_data buffer has been sent to the land station.

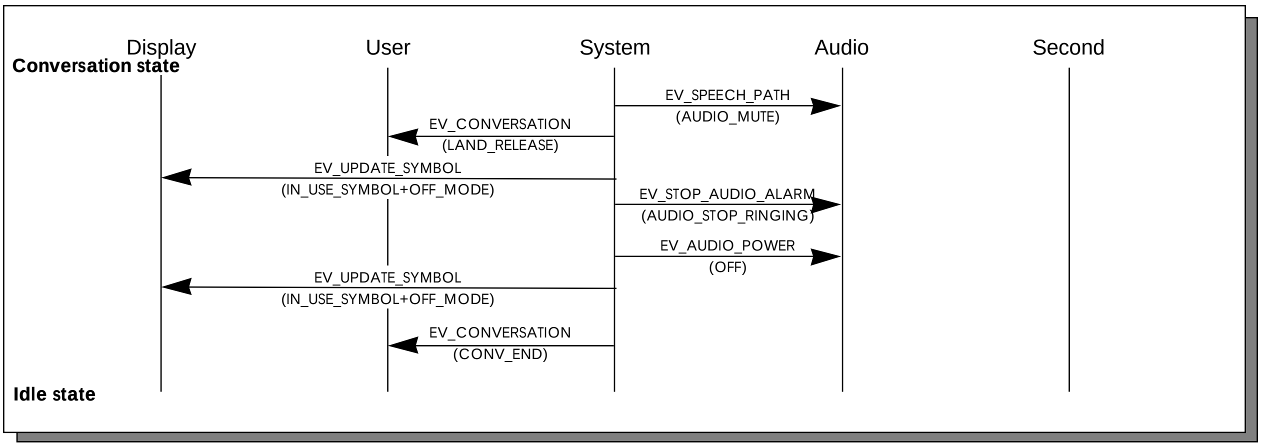
Knock On

Ringing Call / Silent Call

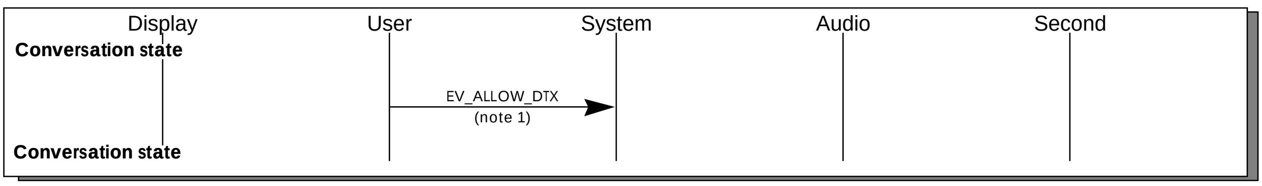


(1) In case of Silent Call this message is not sent.
(2) When Silent Call this status is SILENT_CALL.

Land Release

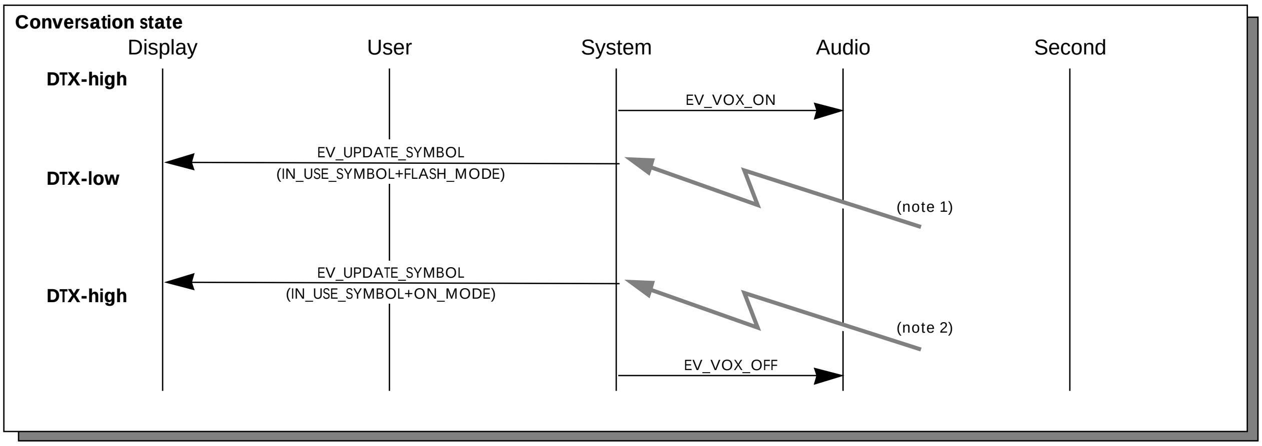


Change DTX Setting



(1) dtx_setting is either INHIBIT_DTX or ALLOW_DTX.

Discontinuous Transmission

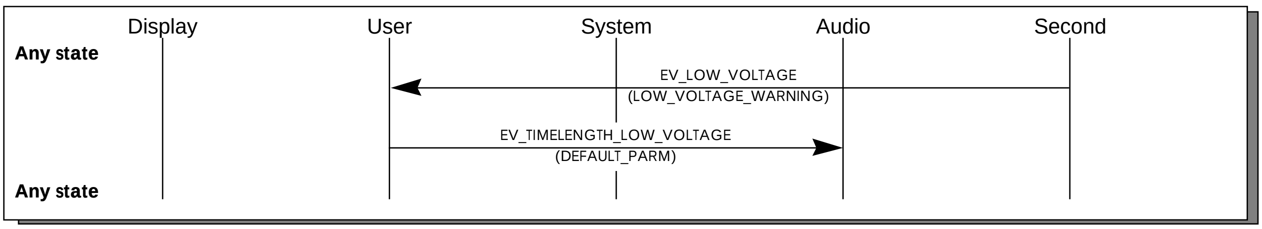


(1) Reason to go to DTX-low has occurred. Reasons are a.o. dtx holdoff period expired and silence detected.
(2) Reason to go to DTX-high has occurred. Reasons are a.o. mobile needs to send something and voice detected.

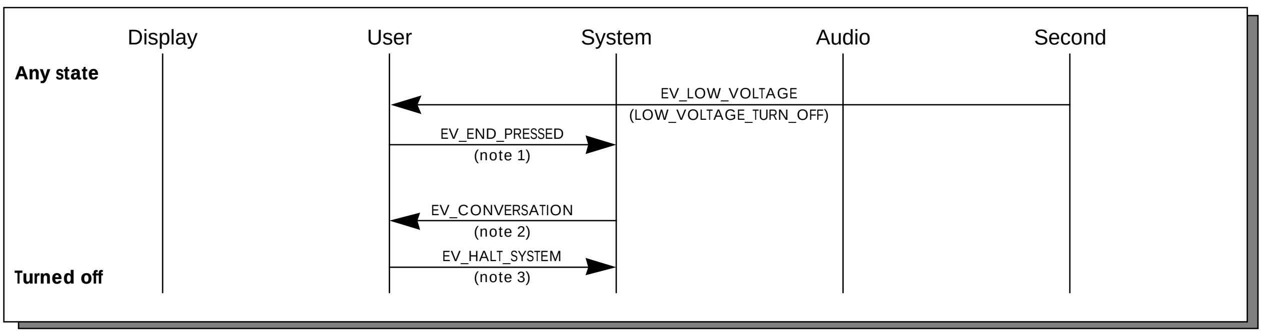
5.3.6 MMI

Battery Low

Battery below Warning Level

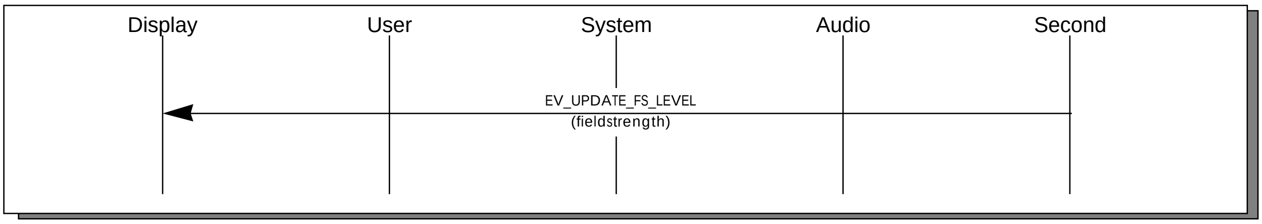


Battery below Turn Off Level

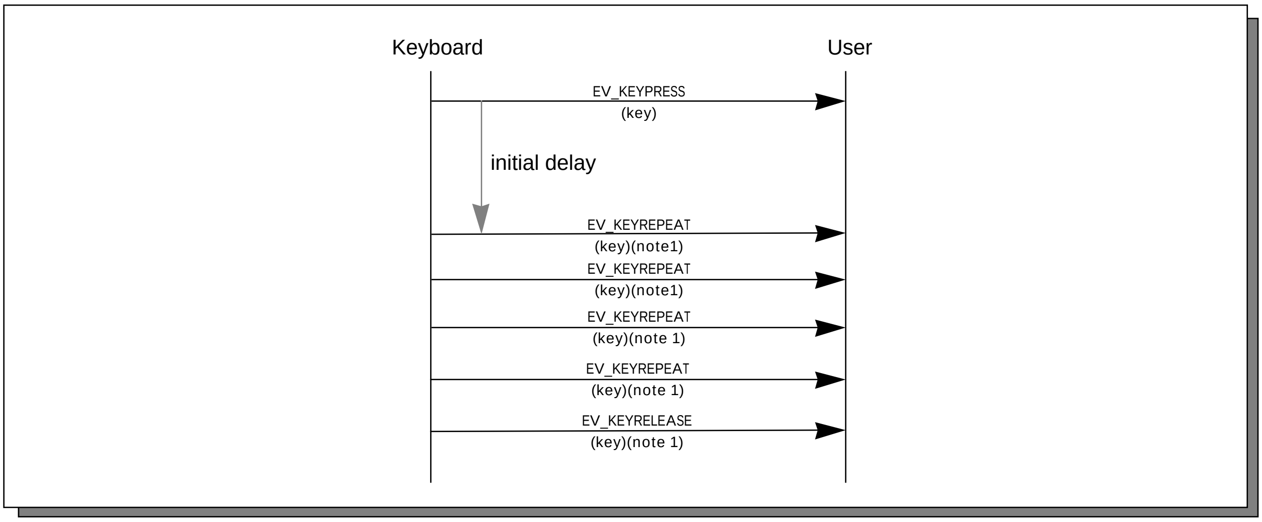


(1) This message is only to be sent when the mobile is in conversation state. After sending this message, the User task must wait for the EV_CONVERSATION message before sending the EV_HALT_SYSTEM message.
(2) This message is only to be expected when the mobile is in conversation state. The value in the message can be CONV_END or NO_CONNECT.
(3) After this message is sent, the User task can turn off the mobile.

RSSI level



Keypress



(1) These messages are sent for the keys that have automoatic key repeat.

5.4 Software Drivers

Software drivers are added to enable tasks to correctly address the different hardware components. The software drivers are shown in figure 4. This figure shows which drivers are called by the OM4754 software and/or by other drivers.

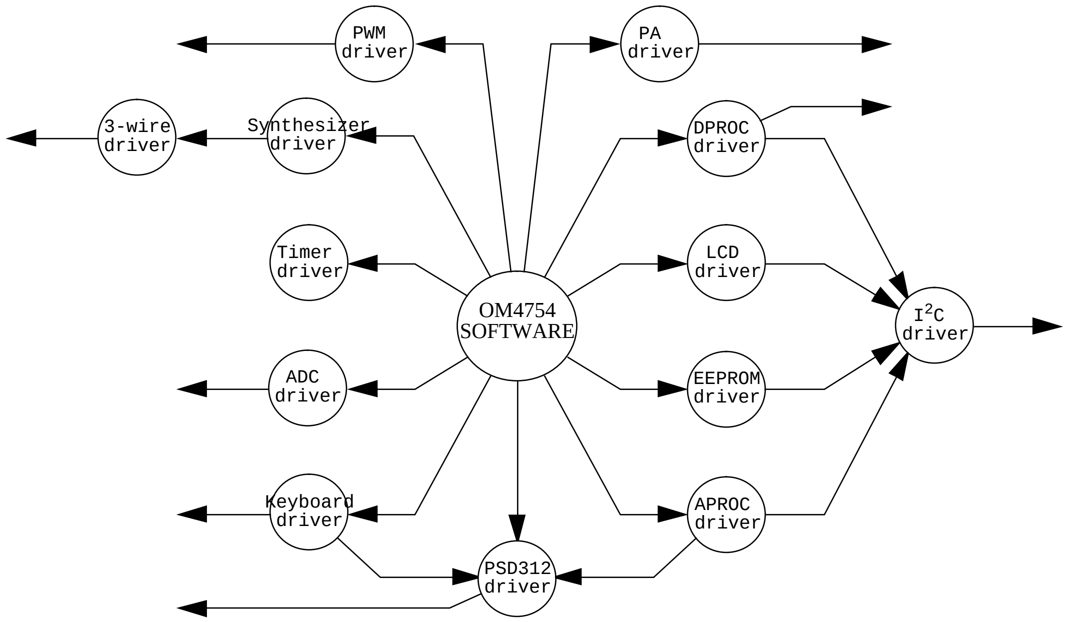


Fig.4 OM4754 driver software

Drivers are implemented as routines to be called by the tasks and/or as interrupt routines. The tasks and interrupt routines communicate through mailboxes and global variables. In the next sections all relevant drivers are described.

5.4.1 The APROC driver

The APROC driver can be found in the files `aproc.c`, `aproc.h` and `vox_int.c`. The SA5753 APROC is controlled through 9 registers. These registers can only be written to via the I^{2}C bus. To simulate a read on these registers the values written via I^{2}C are also saved in shadow registers. There are several routines available to control specific functions in the APROC. All the functions can be controlled by (re)setting appropriate bits in the registers. Because more than one function can be controlled by one register it is necessary to first read the current value from the register, set the appropriate bits and then write the resulting value back in to the register. Only 2 functions in the file `aproc.c` are used by the EAMPS software. These 2 functions are described below, all other functions are not but can be used. To use a function the compilation switch `TEST_HARNESS` has to be set. The use of the compilation `TEST_HARNESS` switch is here only used to limit the amount of code in the application and to give the user some idea of how such a routine can be programmed.

The file `aproc.c` contains 2 routines which are used by the EAMPS software these are:

data_io_aproc	This routine has four parameters. The first parameter identifies the command given. The command is either `AP_READ` for read, `AP_WRITE` for write, `AP_INIT` to initialize APROC or `AP_RESTORE` to restore the values of the shadow registers into APROC. The second parameter is the first register to read from or to write to. The third parameter identifies the number of registers that has to be read/written. The last parameter is the buffer in which the registers are/must be stored. For the commands `AP_INIT` and `AP_RESTORE` the whole APROC register contents is updated and all other parameters are ignored.

aproc_set_vox	This routine has one parameter which identifies whether the VOX should be switched on or off. The VOX is controlled by the VOXctl bit of the SA5753. First this routine disables the VOX interrupt to avoid glitches, then the VOXctl bit is switched on/off and when the VOX is switched on, the VOX interrupt polarity is set and the VOX interrupt is enabled. The VOX interrupt polarity is set according the VOXOUT signal which goes from the SA5752 pin 5 to port 1 bit 1 (P1.1) of the micro controller. If VOXOUT is high the interrupt will be generated on a falling edge (low polarity) otherwise on a rising edge (high polarity).

The VOX interrupt routine can be found in the file `vox_int.c`. The VOX interrupt is connected to port 1 bit 1 (P1.1) of the micro controller, therefore the VOX interrupt is identified as external interrupt 3. When a VOX interrupt is generated the `vox_int()` interrupt routine is activated. This routine checks the polarity of the VOXOUT signal and sends the event EV_VOICE_DETECT with the indication VOICE_DETECTED for a high polarity interrupt and SILENCE_DETECTED for a low polarity interrupt. Then the polarity is changed from low to high or from high to low. In figure 5 the VOXOUT signal, the indication of the event EV_VOICE_DETECT and the polarity are given.

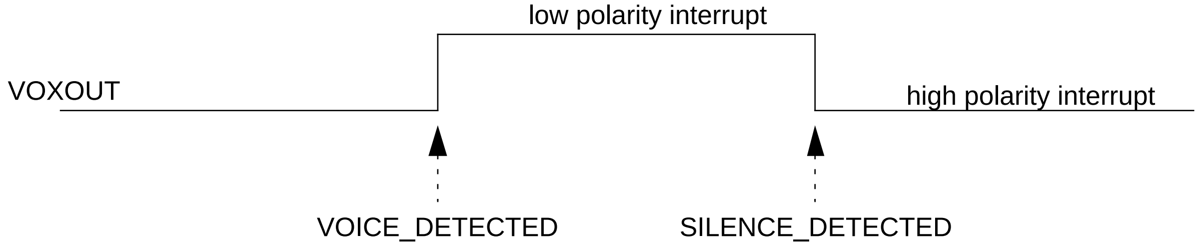


Fig.5 The VOXOUT signal

5.4.2 The DPROC driver

The UMA1000LT DPROC driver can be found in the files `dproc.c`, `dproc.h` and `dprocint.c`. The DPROC transmits and receives frames in AMPS or TACS signalling format. Frames can be in control channel format or voice channel format. Frames received are read from the DPROC using two lines (RXLINE and RXCLK). Frames to be sent are written to the DPROC using two other lines (TXLINE and TXCLK). The data is clocked into or out off the DPROC. The data processor has a status and a control register that can be accessed via the I^2C bus. The status register can be read and the control register can be written. The value written to the control register is saved in a shadow register to make it possible to control only one of the functions in the control register while leaving the other functions unchanged. The following routines can be used to control DPROC:

dproc_send_dummy_frame	This routine performs a TXRESET on DPROC and sends a dummy frame to it. After the dummy frame is sent a second TXRESET is performed to ensure that DPROC is in a defined state.
init_dproc_driver	This routine is called from the Start-up task to reset DPROC and to initialize the DPROC driver. A DPROC reset is given by pulsing NRES. The DPROC is set via I^2C to support the AMPS protocol and then a dummy frame is sent by calling the routine `dproc_send_dummy_frame()`.
enable_dproc_rx	This routine enables the DPROC receive interrupt routine.
disable_dproc_rx	This routine disables the DPROC receive interrupt routine.
send_frame	This routine sends one frame which is given as a parameter. The type of the frame is `TX_FRAME` which is defined in the file `dproc.h`.
wait_for_tx_ready	This routine waits until the next frame can be sent to DPROC. The next frame can be sent when the TXLINE goes high.

wait_for_tx_complete	This routine waits until a transmission is completed. A transmission is completed when bit 2 (TXIP) of the DPROC I^2C status register is set to 0.
dproc_control	With this routine the DPROC control register is programmed. This routine has two parameters, the first indicates which bits must be set and the second parameter indicates which bits must be reset. The values and names of the DPROC control register bits are defined in the file `dproc.h`. An I^2C transmission is only done when the control register is changed. This routine returns the new value of the control register when the I^2C is completed.
dproc_status	With this routine the DPROC status register is read. The parameter given identifies which status register bit(s) should be checked.The values and names of the DPROC status register bits are defined in the file `dproc.h`. This routine returns the value of the specified bit(s) in the status register.

Frames received by the DPROC must be processed quickly. When the DPROC receives a frame it pulls down the received data line (RXLINE). The RXLINE is connected to port 1 bit 5 (P1.5) of the micro controller, therefore the DPROC interrupt is identified as external interrupt 5. The RXLINE causes an interrupt that activates the DPROC interrupt routine `dproc_rx_int()`. This routine clocks out the received frame, stores it at designated memory location, sends a message to the system task indicating a frame is ready to be processed. The interrupt routine checks if the mobile is on a voice or a control channel and calls the routine `check_FVC_frame()` or `check_FOCC_frame()` respectively, to determine the frame type. The DPROC interrupt routine and the routines described above can be found in the file `dprocint.c`.

5.4.3 The Synthesizer driver

The Synthesizer driver can be found in the files `synt1015.c` and `synt.h`. The first IF frequency used is at 86.85 MHz, the channel spacing is 15 kHz, synthesizer A is the transmit synthesizer and synthesizer B is the receive synthesizer.

The synthesizer driver controls the UMA1015M dual synthesizer that is used to tune the mobile to the correct receive and transmit frequencies. The synthesizer driver converts the channel numbers used into divider values and loads these into the synthesizer via a three wire control interface. When the mobile is not transmitting, the transmit synthesizer can be turned off to save the battery. The synthesizer driver is initialized when executing the macro `SYNT_INIT()`. This macro calls the routines `synt_init_test()`, `synt_load_config()`, `synt_load_ref()` and `synt_powerdown_tx() respectively`. The following routines can be found in the file `synt1015.c`:

synt_init_test	This routine initializes the UMA1015M test register to avoid unwanted values in the test register due to power up. This routine is called at system start-up.
synt_powerdown_tx	This routine puts the UMA1015M transmit synthesizer in power down mode and port P3 of the UMA1015M is set to disable the transmit VCO and the power control loop. It also reset's the flag `synt_config_loaded_flag` to indicate that the configuration register of the UMA1015M has to be loaded when a transmit channel is programmed.
synt_load_config	This routine loads the configuration register of the UMA1015M synthesizer and sets the flag `synt_config_loaded_flag` to 1. The transmitter is now powered up and ready for use.
synt_load_ref	This routine loads the reference divider of the UMA1015M synthesizer and sets the flag `synt_ref_loaded_flag` to 1.
synt_load_rx	This routine loads the reference divider if it was not loaded and then programs the receive synthesizer to the desired channel. The channel number should be given as a parameter.

synt_load_tx	This routine loads the configuration register and the reference divider if they were not loaded and then programs the transmit synthesizer to the desired channel. The channel number should be given as a parameter.

5.4.4 The Timer driver

The Timer driver can be found in the files `timer.c`, `timer.h` and `timerint.c`. The timer driver supplies slow timers and fast timers. A fast timer can be set in increments of 1 ms, a slow timer in increments of 20 ms. When a timer is started, the driver puts information about the timer in a global structure. A timer handle that identifies the timer is returned. The following routines and macro's are present in the files `timer.c` and `timer.h`:

init_timers	This routine initializes the amount of fast and slow timers, then hardware timer 1 and 2 and their corresponding interrupt routines are initialized. This routine is called from the Start-up task.
start_fast_timer	This macro calls the routine `start_a_fast_timer()` with the last parameter set to ONE_SHOT. The two parameters given are supplied as the first two parameters to the routine `start_a_fast_timer()`.
start_fast_cont_timer	This macro calls the routine `start_a_fast_timer()` with the last parameter set to RELOADABLE. The two parameters given are supplied as the first two parameters to the routine `start_a_fast_timer()`.
start_a_fast_timer	This routine is called with 3 parameters. The first parameter is the time in milli seconds which the timer should run, the second is the mailbox where the message should be sent to when the timer expires. If the mailbox is NO_MBX a signal is sent to the invoking task when the timer expires. The third parameter is either ONE_SHOT or RELOADABLE. If the timer could not be started 0 is returned otherwise the timer handle is returned.
start_slow_timer	This macro calls the routine `start_a_slow_timer()` with the last parameter set to ONE_SHOT. The two parameters given are supplied as the first two parameters to the routine `start_a_slow_timer()`.
start_slow_cont_timer	This macro calls the routine `start_a_slow_timer()` with the last parameter set to RELOADABLE. The two parameters given are supplied as the first two parameters to the routine `start_a_slow_timer()`.
start_a_slow_timer	This routine is called with 3 parameters. The first parameter is the number of 20 milli second intervals for which the timer should run, the second is the mailbox where the message should be sent to when the timer expires. If the mailbox is NO_MBX a signal is sent to the invoking task when the timer expires. The third parameter is either ONE_SHOT or RELOADABLE. If the timer could not be started 0 is returned otherwise the timer handle is returned.
reload_slow_timer	This routine reloads a slow timer. As parameters the time, the mailbox and the timer handle are given. If the timer was not found the timer is started for the given period. If the timer was found and the remaining delay is greater than the specified time nothing will be done, otherwise the timer is reloaded with the new specified time. The timer handle of the (new) timer is always returned.
stop_timer	This macro calls the routine `stop_a_timer()` and sets the timer handle to 0 so that the timer handle is forgotten. It is better to use this macro instead of using the routine `stop_a_timer()`.
stop_a_timer	This routine stops the timer identified by the given timer handle. If the timer is unknown ERROR is returned otherwise SUCCESS will be returned.

stop_timer_category	This routine stops all timers that the invoking task has running when the first parameter is STOP_TSK, otherwise (first parameter is STOP_MBX) it stops all running timers which send a message to the mailbox identified by the second parameter. In either way the number of timers that have been stopped will be returned.
restart_timer	This routine restarts any fast or slow timer identified by the parameter timer handle. If the timer was restarted it returns SUCCESS otherwise it returns ERROR.

The timer interrupt routines go through the global timer structures and decrease the remaining time till expiration. When a timer expires a message or a signal is sent to the mailbox/task specified for that timer. A message is sent when the mailbox identifier is not equal NO_MBX, otherwise a signal is sent. When the timer is a reloadable timer it is started again. The timer interrupt routine can be found in the file `timerint.c`. The fast 1 milli second timer uses timer 2, the slow 20 milli second timer uses timer 1. Timer 0 is used for the system clock of the RTX-51 operating system.

In Table 11 is a list of timers which can be used simultaneously by a tasks, which does not mean that these timers are actually in use. In order to see the maximum amount of timers in use the variables `max_fast` for all fast timers and `max_slow` for all slow timers can be checked.

TABLE 11 Maximum number of timers simultaneously used by a task

Task	Number of Slow timers	Number of Fast timers
Idle task	0	0
Display task	1	1
Second task	1	0
Keyboard task	1	0
Start-up task	0	0
Audio task	1	1
User task	1	1
System task	7	3

Currently the maximum number of slow timers is set to 15 and the maximum number of fast timers is set to 10 (see `timer.h`).

5.4.5 The EEPROM driver

The EEPROM driver can be found in the files `eeprom.c` and `eeprom.h`. The EEPROM consists of pages which are 16 (EE_PAGE_SIZE) bytes long. The EEPROM contents is kept in RAM to speed up an EEPROM read access because the EEPROM is connected to the I^2C bus. At system start-up the EEPROM contents is read and stored in RAM. All read actions on the EEPROM are done in RAM, all write actions are first written in RAM and then the corresponding EEPROM write action is done. It is also possible that the EEPROM contents is written back to EEPROM at the moment the system is switched off but this implementation is not chosen to minimize the chance of data loss in case of a battery disconnection. The EEPROM structure `eeprom_shadow` is defined in the file `eeprom.h` and is a union of an array `shadow_array` which has the length of the EEPROM and a structure `shadow` in which the appropriate EEPROM variables are defined. In this structure (`shadow`) the variable `ETACS_system_used` defines if the EEPROM was initialized for EAMPS or ETACS. When this variable equals 1 the EEPROM was initialized for ETACS, when it equals 0 the EEPROM was initialized for EAMPS. The first byte of the EEPROM identifies the state of the EEPROM. When this byte contains 0xAA the EEPROM was initialized, when this byte contains 0x55 the EEPROM driver is busy writing to the EEPROM. Any other value of this byte indicate that the EEPROM is empty and needs to be initialized. The following routines are available in the EEPROM driver:

eeprom_read_bytes	This routine performs the actual read on the EEPROM. First the read address is set and than the bytes are read. This routine is called from the two routines below.
eeprom_read_page	This routine reads one complete page from EEPROM to the shadow area in RAM.
eeprom_read_to_shadow	This routine reads the whole EEPROM contents to the RAM shadow area. If the variable `ETACS_system_used` equals 1 or the first byte in EEPROM indicates that the EEPROM is empty the RAM shadow area is cleared, some EEPROM variables are initialized and the whole shadow area is written back to the EEPROM. At the end of this routine the EEPROM check-sum is also checked.
eeprom_write_bytes	This routine performs the actual write to the EEPROM. During this routine the I^2C bus is enabled. First the first byte in EEPROM is set to write in progress then the bytes are written, a new check-sum is calculated and stored in EEPROM and then the first byte in EEPROM is set to normal operation. This routine is called from one of the four routines below.
eeprom_write_page	This routine writes one page from the RAM shadow area to the EEPROM.
eeprom_write_sys_pages	This routine writes one system page from the RAM shadow area to the EEPROM. The system pages are the first 8 (SYS_SIZE) pages in EEPROM.
eeprom_write_tel_no	This routine writes the telephone number from the RAM shadow area to the EEPROM. The telephone numbers are stored after the system pages until the end of the EEPROM.
eeprom_write_last_no_pages	This routine writes the last number dialled pages from the RAM shadow area to the EEPROM. The last number pages are currently only in RAM, however in the EEPROM driver a provision is made to be able to store the last dialled numbers in EEPROM.

All EEPROM routines return after the EEPROM access is completed.

5.4.6 The I^2C driver

The I^2C driver can be found in the files `iic.c`, `iic.h` and `iic_int.c`. The I^2C driver controls the I^2C hardware that is part of the microcontroller. It provides functions to read and write data from and to the I^2C bus. Writing data to another I^2C device on the I^2C bus is done by sending the I^2C address with a write indication and then writing the data on the bus. Writing data to another I^2C device in this way is called a master transmit. Reading data from another I^2C device in the I^2C bus is done by sending the I^2C address with a read indication and then reading the data written on the bus by the addressed I^2C device. Reading data from another device in this way is a called master receive.

Data to be written to the I^2C bus is put in a structure. When the I^2C hardware is not busy transmitting previous data on the I^2C bus a start condition is initiated. When the start condition has been sent an interrupt is generated. The I^2C interrupt triggers the I^2C interrupt routine. This routine checks the status of the I^2C hardware and takes appropriate action. When another data byte is to be sent or received, a start or stop condition is initiated, and the data is sent or received. When a complete stream of data has been sent/received, a message is sent to the mailbox specified in the global structure by the task that wanted to send or receive the data.

The I^2C interrupt routine is also programmed to react on slave receive mode. This means that the I^2C driver supports multiple masters on the I^2C bus. The I^2C address of the 83CL580 micro controller is set to 6 in the file `iic.h`. When a slave receive is completed the data is stored in a global buffer and a message is sent to the IIC_SR_MBX mailbox. Please note that the IIC_SR_MBX is also used by the User task to synchronize with the EEPROM driver (see Table 8). The slave receive part of the I^2C driver is not used by this application and is also not thoroughly tested. However when using the slave receive mode the I^2C bus must be enabled. When the files `iic.c` and `iic_int.c` are compiled with the compiler switch `PRODUCTION_PHONE` set the I^2C bus will be disabled after an I^2C transmission by pulling the SCL line low. If the I^2C bus must be enabled the routine `iic_bus_enable()` must be called. When using an emulator the variable `iic_disable` can be set to `0`.

In the file `iic.h` the I^2C slave addresses are defined. In this file the macro `IIC_SETUP()` is defined. This macro is called from the file `p83cl580.c` and initializes the I^2C hardware. The following routines are available in the file `iic.c`:

iic_module_init	This routine initializes the I^2C driver. This routine does not initializes the I^2C hardware, this is done by the macro `IIC_SETUP()`. During this routine all interrupts are disabled.
iic_bus_enable	This routine enables the I^2C bus so that an outside party can access the I^2C bus.
iic_bus_disable	This routine disables the I^2C bus so that no outside party can access the I^2C bus.
set_iic_SR_task	This routine tells the I^2C interrupt routine which task will receive a message when data is received in slave receive mode.
iic_put_byte	This routine puts one byte on the I^2C bus. This routine has three parameters, the first is the slave address of the I^2C device which should be accessed, the second is the mailbox to which the confirmation should be sent to. When this parameter equals NO_MBX no confirmation is sent. The third parameter is the byte which should be sent.
iic_put_string	This routine puts a complete string on the I^2C bus. This routine has four parameters, the first is the slave address of the I^2C device which should be accessed, the second is the mailbox to which the confirmation should be sent to. When this parameter equals NO_MBX no confirmation is sent. The third parameter is the pointer to the string which should be sent and the fourth parameter is the number of bytes to be sent.
iic_read_byte	This routine reads one byte from the I^2C bus. This routine has three parameters, the first is the slave address of the I^2C device which should be accessed, the second is the mailbox to which the confirmation should be sent to. When this parameter equals NO_MBX no confirmation is sent. The third parameter is the pointer to the byte which should be read. Please note that when reading from the I^2C bus the calling routine must always wait for confirmation to be sure that the data is valid.
iic_read_string	This routine reads a complete string from the I^2C bus. This routine has four parameters, the first is the slave address of the I^2C device which should be accessed, the second is the mailbox to which the confirmation should be sent to. When this parameter equals NO_MBX no confirmation is sent. The third parameter is the pointer to the string which should be read and the fourth parameter is the number of bytes which should be read. Please note that when reading from the I^2C bus the calling routine must always wait for confirmation to be sure that the data is valid.

5.4.7 The Keyboard driver

The Keyboard driver can be found in the file `kb_tsk.c`. The keyboard is connected to port 4 of the micro controller, except for the ON/OFF key which is connected to the PSD312L. On order to read a key from the keyboard the routine `get_key()` must be called which returns the (debounced) key pressed. All keys have a higher priority than the ON/OFF key, which means that when a key is pressed together with the ON/OFF key the key pressed is processed, but when two other keys are pressed together no key is processed. When a key is pressed and stays pressed the key is only seen once, only the CLEAR and ON/OFF key are repeated. The CLEAR key is repeated when pressed longer than 400 milli seconds, the ON/OFF key is repeated as soon as it stays pressed (and no other key is pressed). The keyboard driver contains the next routines:

kboard_scan	This is the actual keyboard scanning routine. It scans port 4 of the micro controller to see if a key was pressed, if no key was pressed the ON/OFF key is checked.
debounce	This routine debounces a key read from the keyboard by checking it again the next time this routine is called. The keyboard is read by calling the routine `kboard_scan()`.
convert_key	This routine converts a key read from the keyboard into a key number used by the mailbox software.
get_key	This routine gets a key number from the keyboard having debounced it. It calls the above routines in the order listed.

5.4.8 The LCD driver

The LCD driver can be found in the file `lcd_drv.c`. The LCD used is a LP3800-A, which has 3 lines of 12 characters. Line 1 and 3 are used for status information, line 2 is used to display telephone numbers. In figure 6 the LCD layout shown.

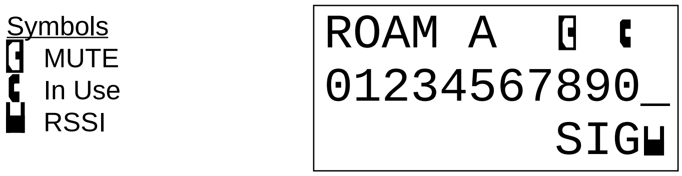


Fig.6 LCD layout

In figure 6 the mobile is roaming on system A, because the text “ROAM A” is displayed, for system B the A is changed in B. The text ROAM can change to NSVC when the mobile is out of service or when the mobile is in its home area A or B is displayed. In conversation mode the In Use symbol will appear on the LCD, when the In Use symbol starts flashing the DTX mode is entered. The RSSI symbol is a bucket which can be filled, when the bucket is full a strong RSSI is measured, when it is empty almost no RSSI is measured. The RSSI is not displayed when the mobile is out of service, hence when the text NSVC is displayed on the first line. The text SIG is put in front of the RSSI symbol to indicate that it is the signal strength and not e.g. the battery level. The MUTE symbol will appear when in conversation the mute key is pressed, however currently the keyboard does not support a mute key. The LCD driver contains the following routines:

Display_Init	This function initializes the LP3800-A LCD display to function set 8 bit, 4 lines Vgen off, display on, cursor off, blink off, increment entry mode, no shift and clears the display.
Display_Update_Line1	This function updates line 1 of the LCD with the data stored in the global array `display_prompt_line`. Any symbol displayed on this line is hidden, if the global array is empty the symbols are restored in this line.
Display_Update_Line2	This function updates line 2 of the LCD with the data stored in the global array `display_num_line`. If the global array is empty nothing is displayed.
Display_Icon	This routine has 2 parameters. The first parameter identifies the icon, the second the RSSI level. Icons are divided in 2 nibbles, one nibble is either OFF_MODE, ON_MODE, FLASH_MODE or FLASH_REVERSE_MODE. The other nibble

identifies the symbol as shown in table 12. The RSSI level is only updated if the icon type is SIG_SYMBOL ored with ON_MODE.

TABLE 12 Symbol types

Symbol name	Meaning
ROAM_SYMBOL	The mobile is roaming
NO_SYMBOL	No service found
SERV_SYMBOL	Service found
IN_USE_SYMBOL	The mobile is in conversation mode
LOCK_SYMBOL	The mobile is locked (not implemented)
A_SYMBOL	The mobile is currently switched to system A
B_SYMBOL	The mobile is currently switched to system B
VOX_SYMBOL	The VOX is switched on (not implemented)
DTMF_SYMBOL	The DTMF dialling is enabled (not implemented)
FUNC_SYMBOL	The function mode is entered (not implemented)
ALPHA_SYMBOL	The alpha mode is entered (not implemented)
MUTE_SYMBOL	The mobile is muted
SIG_SYMBOL	The RSSI must be updated
RECALL_SYMBOL	A memory recall is done (not implemented)
STORE_SYMBOL	A memory store is done (not implemented)

5.4.9 The 3-wire driver

The 3-wire driver can be found in the files `3wire.c` and `3wire.h`. The 3-wire interface is a write only interface that uses an enable line, a clock line and a data line. The 21 data bits are output on the data line clocked by the clock line. The enable line indicates when valid data is present. The 3-wire driver contains one routine:

three_wire_write — This routine performs a complete write on the 3-wire bus. The length of a message is 21 bits which means that the message has a length of 3 bytes. The first 3 bits of the message are ignored. The message is given as a parameter.

5.4.10 The PSD312L driver

The PSD312L driver can be found in the files `psd312.c` and `psd312.h`. The PSD312L is a single chip programmable peripheral. Its offers additional 64k bytes ROM, 2k bytes RAM and 16 I/O ports. The PSD312L driver controls the I/O ports on the PSD312L device. Several outputs and one input are connected to the PSD312L on the mobile. The following routines can be used to access the PSD312L driver:

psd312_init — This routine initializes the data direction register of port A to 1, since port A is programmed to track A0 until A7. Then it initializes the data register B and data direction register B. Bit 1 of port B (PB1) is used for input, all other bits of port B are set for output. The data register is programmed before the direction register to avoid glitches due to previous data.

psd312_read — This routine reads the I/O port and returns it value to the calling process. An input port is read from the pin register and an output port is read from the data register. The I/O port is identified by the parameter given, the definitions of all I/O ports are given in the file `psd312.h`.

psd312_write	This routine writes to the corresponding I/O port. The first parameter identifies the action (SET or RESET), the second parameter identifies the I/O port, the definitions of the I/O ports are given in the file `psd312.h`.

5.4.11 The ADC driver

The ADC driver can be found in the library `amps.lib` and is therefore not delivered in source. Channel 0 of the ADC must therefore always be connected to the RSSI output of the RF system. The ADC driver consists of one routine, the prototype of this routine can be found in the file `stateutl.h`.:

read_ad_converter	This routine starts a conversion on a given ADC channel. The ADC channel (0, 1, 2, or 3) is identified by the given parameter. The ADC value read is returned.

5.4.12 General utilities

All general utilities can be found in the files `io_utl.c` and `io_utl.h`. These files contain routines for turning the power amplifier, the transmit synthesizer and the back lighting on or off and one routine to set the PWM power level, these routine are:

tx_syn_on	This routine powers up the transmit synthesizer by calling the routine `synt_load_config()` of the synthesizer driver.
tx_syn_off	This routine powers down the transmit synthesizer by calling the routine `synt_powerdown_tx()` of the synthesizer driver.
pa_on	This routine switches the power amplifier on by setting the TXDIS line (micro controller port 1 bit 2) to 1.
pa_off	This routine switches the power amplifier off by setting the TXDIS line (micro controller port 1 bit 2) to 0.
light_on	This routine switches the back lighting on by setting the LED_ENABLE line connected to the PSD312L device by calling the routine `psd312_write()`.
light_off	This routine switches the back lighting off by resetting the LED_ENABLE line connected to the PSD312L device by calling the routine `psd312_write()`.
tx_power_level	This routine sets the transmitter power to the level indicated by the parameter `level`. The PMW power table used is set in EEPROM and resides in the array `mid_power_lev`. When an RF system is used which uses more than one PWM power table the variable `channel_number` (see `sysvar.h`) can be used to determine the power table to be used.