Title	**UPS 시리얼 프로그래밍 가이드**
No.	

Rev.	Date	By	History
0.0.1	2003-04-09	UIS	Initial draft
0.0.7	2003-05-19	KCY	Title of this document, Chapter 2.1, small changes
0.0.9	2003-07-09	UIS	Easytrap With UPSLink100 2.2.1 upgrade

목차

1. 개요

UPSLink는 기본적으로 세나테크놀로지에서 정의한 UPS와의 통신 규약에 따라 UPS를 감시/제어 하도록 제작되어 있습니다. 하지만 이외에도 사용자가 정의한 UPS 시리얼 통신 규약을 구현한 별도의 사용자 프로그램을 다운로드 받아서 UPS를 감시/제어할 수 있도록 설계되어 있습니다.

세나테크놀로지는 이 사용자 정의 감시/제어 프로그램을 사용자가 최소의 노력으로 간단히 제작이 가능하도록 소스코드와 라이브러리 형태의 여러 가지 도구를 제공합니다. 또한 UPSLink에 기본적으로 내장되어 있는 UPS 시리얼 통신 규약의 정의와 이 규약을 구현한 UPS 시리얼 프로그램의 소스코드를 예제로 제공합니다.

UPSLink는 RFC1628(UPS MIB, UPS의 네트워크 관리를 위한 명세)을 완벽하게 구현하고 있으며, 이 외에도 다양한 기능들로 네트워크를 통한 UPS의 관리를 용이하게 해 줍니다. 단지 시리얼 프로토콜의 구성 및 해석하는 부분만 수정함으로써 UPSLink가 기본적으로 제공하는 모든 기능들을 그대로 유지하면서도 사용자 고유의 UPS의 네트워크 관리를 위한 SNMP 에이전트를 만들 수 있습니다.

이 외에도 간단한 조작을 통해 사용자 경보(SNMP 트랩 포함), 테스트, UPS 감시/설정/제어 내역을 추가하거나 관리용 웹 페이지의 내용 조차도 추가/변경이 가능합니다.

1.1. 기본 구조

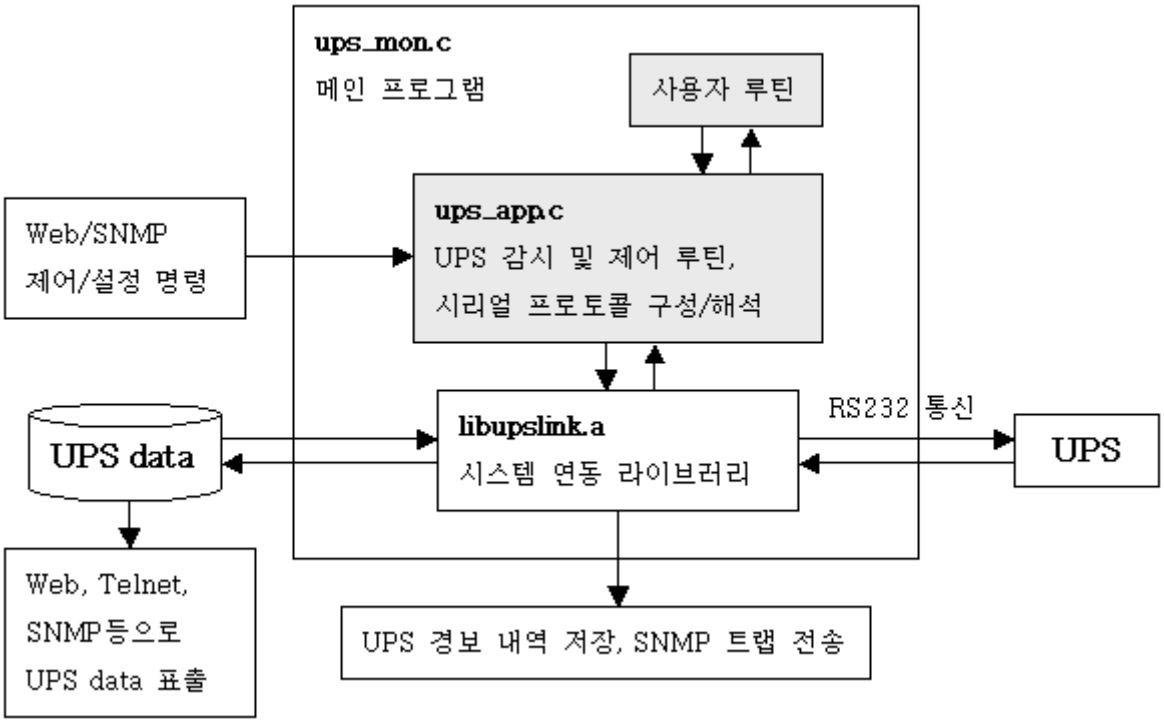

그림 1-1 UPS 시리얼 프로그램 구조도

그림 1-1에서 음영 표시된 부분은 사용자가 수정하거나 추가해야 할 부분입니다. 이 부분들을 제외한 나머지는 소스코드와 라이브러리 형태로 제공됩니다.

1.2. 구성 요소

기본적으로 사용자가 수정해야 하는 파일은 ups_app.c 뿐입니다. 그 외의 것들은 단지 순수하게 또는 함수의 원형을 확인하기 위해 참고하기만 하면 됩니다.

ups_mon.c

UPS 시리얼 프로그램의 메인 함수가 있는 파일, 프로그램 초기화, 프로그램 종료, 데이터 초기화, 시리얼 포트 열기/닫기 등의 기본적인 필수 기능들을 수행합니다.

Note : 이 파일은 사용자가 반드시 수정할 필요가 없습니다.

ups_data.h

UPS 데이터 구조체가 선언되어 있습니다. 데이터의 구조는 기본적으로 RFC1628에서 명시된 항목들과 사용자가 동적으로 추가할 수 있는 영역들로 이루어져 있습니다.

Note : 이 파일은 사용자가 수정할 수 없습니다.

ups_alarm.h

UPS의 경보 내역을 정리해서 선언해 둔 파일입니다.

libupslink.a

시리얼 포트를 통해 UPS와 통신하는 대리자 역할을 합니다. ups_app.c에서 해석한 UPS 데이터를 저장영역에 저장하거나 이미 저장된 데이터에 접근이 가능하게 해 줍니다. UPS의 경보 발생시에 경보내역을 기록하고 SNMP트랩이나 이메일을 통해 외부로 그 내용을 전송해 줍니다. 이러한 모든 동작은 자동으로 이루어 지기도 하고, 때에 따라 ups_app.c에서 함수 호출을 통해 이루어 지기도 합니다. 이 함수들은 include 디렉터리에 있는 여러 개의 헤더파일에 나뉘어서 선언되어 있습니다.

Note : 사용자가 링크하여 사용 가능하도록 이미 컴파일 된 정적 라이브러리 형태로 제공됩니다.

libups_data.h, ups_msgq.h, ups_ser.h, ups_sem.h, strqueue.h and ups_easytraplog_msgq.h

libupslink.a의 함수 프로토타입들을 선언해 둔 파일들입니다.

libups_data.h는 UPS데이터 관리, 경보 전송등에 관계되어 있고, ups_msgq.h는 웹이나 SNMP등 외부 명령과 관계되어 있으며, ups_ser.h와 strqueue.h는 시리얼 통신 및 시리얼로 받은 데이터 버퍼와 관계되어 있고, ups_easytraplog_msgq.h는 RFC1628이외의 간단한 사용자 정의 SNMP v1 trap을 전송하는 것과 상관있습니다.

Note : strqueue.h에 선언되어 있는 string queue 관련 함수들은 시리얼 프로토콜이 아스키 형태일 경우 유용하게 사용할 수 있도록 제공되는 유틸리티 입니다. 만약 시리얼 프로토콜이 바이너리 형태라면 strqueue.h의 함수들을 사용할 수 없습니다. 이 경우 템플릿 코드에서 string queue를 사용하지 않고, ups_ser_read(..)함수에서 읽혀진 바이너리 데이터를 바로 사용하도록 수정을 가해야 합니다.

Note : 이 파일들은 사용자가 수정할 수 없습니다.

Note : 이 파일은 사용자가 수정할 수 없습니다.

ups_app.c

UPS와의 통신을 위한 프로토콜의 구성 및 해석을 담당합니다. 주기적으로 UPS를 감시하거나, UPS에서 비동기적으로 올라오는 메시지를 처리하도록 구성할 수 있습니다. 순수히 프로토콜을 구성하고 해석하는 부분만 수정하거나 추가하기만 하면 되도록 제작된 완벽한 템플릿 소스 코드와 예제가 제공됩니다. 필요에 따라 별도의 C파일 만을 만들어 이것에 링크 시킬 수도 있습니다.

별도의 프로세스를 하나 더 생성해서 그곳에서는 SNMP나 웹을 통한 사용자의 명령을 받아 처리합니다. 별도의 프로세스에서 시리얼 포트에 각각 접근하지만 멀티 쓰레드가 동시에 사용해도 안전하도록 (thread-safe) 설계되어 있으므로 신경 쓰지 않아도 됩니다.

Note : 이 파일은 사용자가 반드시 수정해야 합니다.

2. 사용자 응용프로그램 제작

2.1. 준비

아래와 같은 순서로 개발 환경을 구축한다.

1) BIOS가 CD-ROM 부팅을 지원하는 PC 한대를 준비한다. 하드 디스크에 2GB의 여유공간이 있어야 합니다.

2) Redhat 7.0 설치

- BIOS를 CD-ROM부팅으로 설정하고 Redhat 7.0 CD를 CD-ROM에 넣고 PC를 부팅합니다.
- Redhat 설치 과정을 진행합니다.

3) Hardhat 설치 (Hardhat CD의 `docs/HHL-JE-20.pdf` 참조)

- Redhat으로 PC를 부팅 시킨 후 root로 로그인한 후에 Hardhat CD를 CD-ROM에 넣고 아래와 같이 인스톨을 진행합니다.

```
/mnt/cdrom/bin/hhl-host-isntall    [enter]
```

- 진행 중에 설치할 LSP를 물으면 `embeddedplanet-rpxlite`라고 적어 넣습니다.
- Hardhat Linux가 `/opt/hardhat`밑에 설치됩니다.
- 경로를 익스포트 시킵니다.

```
export  PATH=$PATH:/opt/hardhat/devkit/ppc/8xx/bin:/opt/hardhat/host/bin
[enter]
```

4) Template 소스 컴파일

- UPSLink CD에서 `UPS_serial_program_template.tar.gz`를 작업 디렉터리에 복사합니다.
- 압축을 해제합니다.

```
tar –xzvf upslink_app_template.tar.gz [enter]
```

- 내용을 확인합니다.

```
#cd upslink_template [enter]
#ls [enter]
Makefile    include    lib    ups_app.c    ups_mon.c
#ls include [enter]
libups_data.h    strqueue.h    ups_alarm.h    ups_data.h    ups_msgq.h
ups_sem.h        ups_ser.h
#ls lib [enter]
libupslink.a
```

- 처음 상태로 컴파일 해 봅니다.

```
#make [enter]
```

- 결과를 확인합니다.

```
#ls [enter]
Makefile    include    lib    ups_app.c    ups_app.o    ups_mon.c
ups_mon.o   upsapp
```

여기서 만들어진 `upsapp`는 아무런 동작도 하지 않는 기본 응용프로그램 입니다. 이제 다음 절을 참고하여 `ups_app.c`를 수정하고 필요한 사용자 루틴을 추가하여 사용자 고유의 UPS 시리얼 프

로토콜 구현 프로그램을 만듭니다.

2.2. ups_mon.c 둘러보기

이 파일은 반드시 수정할 필요는 없습니다. (필요한 경우 약간의 수정을 가하십시오.)

RS232통신으로 읽은 데이터를 저장할 문자열 버퍼 큐 사이즈를 정의하고 문자열 버퍼 큐를 선언합니다. 사이즈를 조절하거나 시리얼 프로토콜이 바이너리 형태일 경우 삭제할 수도 있습니다.

```
#define QUEUE_SIZE 2048
queue upsq;
```

UPS 시리얼 프로그램을 초기화합니다. 프로세스와 관련된 시그널 핸들러를 등록하고 UPS데이터, 메시지 큐, 세마포어 그리고 문자열 버퍼 큐 등을 초기화 하며, 시리얼 포트를 엽니다. 시리얼 프로토콜이 바이너리 형태일 경우 문자열 버퍼 큐 초기화 루틴을 삭제할 수 있습니다.

```
void init_ups_mon(bool bfirst){
    signal(SIGINT, ups_mon_sighandler);
    signal(SIGTERM, ups_mon_sighandler);
    signal(SIGCHLD, ups_mon_sighandler);

    init_upsdata(bfirst);
    init_upstraplogmsgq();
    init_upseasytraplogmsgq();
    init_upssem();
    init_upsmsgq();

    // init data queue
    if(queue_init(&upsq, QUEUE_SIZE) < 0){
            fprintf(stderr, "queue init failed.. exiting..\n");
            exit(0);
    }
    // open serial device
    if(ups_ser_open()<0)  {
            fprintf(stderr, "Can't open serial device.. exiting..\n");
            exit(0);
    }
}
```

프로세스 관련 시그널 핸들러 입니다. UPS데이터를 정리 하거나 시리얼 포트를 닫는 등 프로그램 종료시에 수행되어야 할 루틴들이 포함되어 있습니다.

```
void ups_mon_sighandler(int sig){
    switch(sig){
            case SIGINT:
            case SIGTERM:
                    ups_ser_close();
                    finalize_upsdata();
                    queue_free(&upsq);
                    exit(0);
            case SIGCHLD:
                    {
                            int status;
                            pid_t pid;

                            pid = waitpid(-1, &status, WNOHANG);
                            if (pid <= 0)
                                    break;
                    }
                    break;
    }
```

```
}
```

메인 함수에서는 ups_app.c에 있는 실제 프로그램인 start_ups_app()를 수행시킵니다.

```
main(){
    printf("Starting ups_mon.. \r\n");
    if(!fork()){
            init_ups_mon(true);
            start_ups_app();
    }
}
```

2.3. UPS 데이터

ups_data.h

ups_data.h는 UPSLink의 메모리 상에 저장될 UPS데이터 구조체들을 정의하고 있습니다. 이 데이터 구조는 RFC1628을 완벽하게 구현 가능하도록 표준에 근거하여 만들어져 있으며, 이에 추가로 별도의 사용자 영역이 선언되어 있습니다.

사용자는 이 데이터에 직접 접근이 불가능하며, libups_data.h에 선언되어 있는 대리 함수를 사용해서 데이터를 읽거나 쓸 수 있습니다. 데이터는 'ID, 배터리, 입력, 출력, 바이패스, 경보, 설정, 제어, 테스트, 기타 그리고 사용자 영역'의 작은 구조체들로 이루어져 있으며 데이터를 읽거나 쓸 때에는 이 구조체 단위로 읽거나 쓰게 됩니다.

ups_alarm.h

ups_alarm.h는 UPS 경보 내역에 대한 인덱스를 정의하고 있습니다. 경보의 발생 및 해제와 관련된 부분을 수정할 경우 참고해야 합니다.

2.4. 라이브러리의 사용 : libupslink.a

UPSLink의 라이브러리인 libupslink.a는 컴파일된 상태로 제공됩니다. 여기에서는 제공되는 함수들의 사용법을 설명합니다.

libups_data.h

libups_data.h에는 UPS 데이터에 사용자가 접근할 수 있도록 하는 함수들이 선언되어 있습니다.

init_upsdata : UPS 데이터를 초기화 합니다. 프로그램 시작 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
void init_upsdata(bool bfirst);
```

reset_upsdata : UPS 데이터 중 ID, 배터리, 입력, 출력 그리고 바이패스 그룹에 해당하는 데이터들을 초기화 합니다. UPS communication lost alarm 과 같은 필요 상황에서 호출해 줍니다.

```
void reset_upsdata();
```

finalize_upsdata : UPS 데이터를 정리합니다. 프로그램 종료 시 ups_mon.c에서 호출되므로

사용자가 별도로 호출할 필요가 없습니다.

```
void finalize_upsdata();
```

경보 그룹

register_usralarm : 사용자 지정 경보를 추가합니다. 이 함수에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usralarm(int usrindex, int oiddepth, unsigned long *oid,
char *shortdesc, char *longdesc);
```

read_usralarm : 해당 usrindex에 등록되어 있는 사용자 지정 경보를 읽습니다.

```
int read_usralarm(int usrindex, struct _usr_alarm_object *alarmobj);
```

set_alarm : 경보가 신규로 발생 또는 발생이 지속되는 상태거나 신규로 해제 또는 해제가 지속되는 경우 호출합니다. 인자 중 force를 false로 하면 함수 내부에서 경보 상태(발생/해제)가 전환되는 경우만 판단을 하여 경보 발생/해제 시각을 기록하고 SNMP 트랩이나 이메일을 전송하므로 이 함수를 사용할 때 경보가 신규로 발생한 것인지 아니면 이미 발생되어 있던 것인지를 신경 쓰지 않아도 됩니다. force를 true로 하면 무조건 기록을 하고 SNMP트랩 및 이메일을 전송합니다. alarmindex는 ups_alarm.h를 참고하여 전달합니다. status는 ALARM_RELEASED와 ALARM_PRESENT중 하나의 값을 전달합니다.

```
int set_alarm(int alarmindex, ALARM_STATUS status, bool force);
```

RFC1628 규정 준수 사항 : set_alarm(..) 함수를 사용하여 신규로 경보를 발생시키면 해당 경보에 대해 고유의 upsAlarmId가 부여됩니다. upsAlarmTestInProgress를 신규로 해제 시켰을 경우 등록된 트랩 수신 서버로 upsTrapTestCompleted 트랩이 전송됩니다. 동시에 upsTestSpinLock값을 증가시켜 SNMP-set 명령을 통한 테스트 수행이 가능한 상태로 설정하며, upsTestElapsedTime이 자동으로 계산되어 저장됩니다. 신규로 발생시킨 upsAlarmTestInProgress와 upsAlarmOnBattery에 대해서는 upsTrapAlarmEntryAdded 트랩을 전송하지 않습니다. 다만 upsAlarmOnBattery가 발생되어 있는 중에는 1분 간격으로 upsTrapOnBattery 트랩을 전송합니다.

사용 예 :

```
set_alarm(_ALARM_COMMLOST, ALARM_PRESENT, false);
```

get_alarmstatus : 특정 경보의 발생/해제 상태를 읽습니다. 반환되는 값은 ALARM_RELEASED와 ALARM_PRESENT중 하나입니다. alarmindex는 ups_alarm.h를 참고하십시오.

```
int get_alarmstatus(int alarmidex);
```

get_present_alarmnum : 현재 발생상태에 있는 경보의 개수를 반환합니다.

```
int get_present_alarmnum();
```

정보 및 상태 그룹

register_usrinfo, read_ups_usrinfo, save_ups_usrinfo : 사용자 지정 정보와 관련된 함수들입니다. 이 함수들에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usrinfo(int usrindex, char *desc, INFO_TYPE type);
int read_ups_usrinfo(int usrindex, int *infoval, char *infostr);
int save_ups_usrinfo(int usrindex, int infoval, char *infostr);
```

register_usrstatus, read_ups_usrstatus, save_ups_usrstatus : 사용자 지정 UPS

상태 정보와 관련된 함수들입니다. 이 함수들에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usrstatus(int usrindex, char *desc, STATUS_TYPE type);
int read_ups_usrstatus(int usrindex, int *statusval, char *statusstr);
int save_ups_usrstatus(int usrindex, int statusval, char *statusstr);
```

read_ups_identification, save_ups_identification : UPS 데이터 영역의 UPS ID정보를 읽거나 씁니다. _UPS_DATA_ID는 ups_data.h에 정의되어 있습니다.

```
int read_ups_identification(_UPS_DATA_ID *data);
int save_ups_identification(_UPS_DATA_ID *data);
```

Note : _UPS_DATA_ID의 agent_sw_ver는 쓰기 제한되어 있습니다.

사용 예 :

```
_UPS_DATA_ID data;
read_ups_identification(&data);
strncpy(data.manufacturer, “My Manufacturer”, sizeof(data.manufacturer)-1);
save_ups_identification(&data);
```

read_ups_battery, save_ups_battery : UPS 데이터 영역의 UPS 배터리 정보를 읽거나 씁니다. _UPS_DATA_BATTERY는 ups_data.h에 정의되어 있습니다

```
int read_ups_battery(_UPS_DATA_BATTERY *data);
int save_ups_battery(_UPS_DATA_BATTERY *data);
```

Note : _UPS_DATA_BATTERY의 각 항목의 저장 단위에 주의하십시오. 예를 들어 voltage와 current는 각각 0.1[VDC]와 0.1[amp DC]입니다. 저장 단위는 ups_data.h에 설명이 되어 있습니다.

read_ups_input, save_ups_input : UPS 데이터 영역의 UPS 입력 정보를 읽거나 씁니다. _UPS_DATA_INPUT는 ups_data.h에 정의되어 있습니다

```
int read_ups_input(_UPS_DATA_INPUT *data);
int save_ups_input(_UPS_DATA_INPUT *data);
```

Note : _UPS_DATA_INPUT의 각 항목의 저장 단위에 주의하십시오. 예를 들어 frequency와 current는 각각 0.1[Hz]와 0.1[RMS amp]입니다. 저장 단위는 ups_data.h에 설명이 되어 있습니다.

read_ups_output, save_ups_output : UPS 데이터 영역의 UPS 출력 정보를 읽거나 씁니다. _UPS_DATA_OUTPUT는 ups_data.h에 정의되어 있습니다

```
int read_ups_output(_UPS_DATA_OUTPUT *data);
int save_ups_output(_UPS_DATA_OUTPUT *data);
```

Note : _UPS_DATA_OUTPUT의 각 항목의 저장 단위에 주의하십시오. 예를 들어 frequency와 current는 각각 0.1[Hz]와 0.1[RMS amp]입니다. 저장 단위는 ups_data.h에 설명이 되어 있습니다.

read_ups_bypass, save_ups_bypass : UPS 데이터 영역의 UPS 바이패스 정보를 읽거나 씁니다. _UPS_DATA_BYPASS는 ups_data.h에 정의되어 있습니다

```
int read_ups_bypass(_UPS_DATA_BYPASS *data);
int save_ups_bypass(_UPS_DATA_BYPASS *data);
```

Note : _UPS_DATA_BYPASS의 각 항목의 저장 단위에 주의하십시오. 예를 들어 current는 0.1[RMS amp]입니다. 저장 단위는 ups_data.h에 설명이 되어 있습니다.

제어 그룹

read_ups_control_shutdowntype, save_ups_control_shutdowntype : UPS의 셧다운 방식을 읽거나 씁니다.

```
int read_ups_control_shutdowntype(SHUTDOWN_TYPE *type);
int save_ups_control_shutdowntype(SHUTDOWN_TYPE type);
```

read_ups_control_autorestart, save_ups_control_autorestart : UPS의 재시작 방식을 읽거나 씁니다.

```
int read_ups_control_autorestart(AUTO_RESTART *type);
int save_ups_control_autorestart(AUTO_RESTART type);
```

save_ups_control_shutdownafterdelay : UPS 예약 셧다운 명령을 전송하면서 이 함수를 호출해야만 웹 페이지의 UPS제어 정보란에 셧다운까지 남은 시간이 올바르게 표시됩니다.

```
int save_ups_control_shutdownafterdelay(long delay);
```

save_ups_control_startupafterdelay : UPS 예약 시작 명령을 전송하면서 이 함수를 호출해야만 웹 페이지의 UPS제어 정보란에 예약 시작까지 남은 시간이 올바르게 표시됩니다.

```
int save_ups_control_startupafterdelay (long delay);
```

save_ups_control_rebootwithduration : UPS 예약 재시작 명령을 전송하면서 이 함수를 호출해야 웹 페이지의 UPS제어 정보란에 재시작까지 남은 시간이 올바르게 표시됩니다.

```
int save_ups_control_rebootwithduration(long duration);
```

register_usrcontrol : 사용자 지정 UPS 제어와 관련된 함수입니다. 이 함수에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usrcontrol(int usrindex, char *desc, CONT_TYPE type, int
defval, char *defstr, char **combo_desc);
```

설정 그룹

read_ups_config, save_ups_config : UPS 데이터 영역의 UPS 설정 정보를 읽거나 씁니다. _UPS_DATA_CONFIG는 ups_data.h에 정의되어 있습니다

```
int read_ups_config(_UPS_DATA_CONFIG *data);
int save_ups_config(_UPS_DATA_CONFIG *data);
```

Note : _UPS_DATA_CONFIG의 각 항목의 저장 단위에 주의하십시오. 예를 들어 frequency와 voltage는 각각 0.1[Hz]와 1[RMS volts]입니다. 저장 단위는 ups_data.h에 설명이 되어 있습니다.

register_usrconfig, read_ups_usrconfig, save_ups_usrconfig : 사용자 지정 UPS 설정과 관련된 함수들입니다. 이 함수들에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usrconfig(int usrindex, char *desc, CONT_TYPE type, char
**combo_desc);
int read_ups_usrconfig(int usrindex, int *configval, char *configstr);
int save_ups_usrconfig(int usrindex, int configval, char *configstr);
```

테스트 그룹

register_usrtest, read_usrtest : 사용자 지정 UPS 테스트와 관련된 함수입니다. 이 함수

에 대한 자세한 내용은 4장을 참고하십시오.

```
int register_usrtest(int usrindex, int oiddepth, unsigned long *oid,
char *testdesc);
int read_usrtest(int usrindex, struct _usr_test_object *testobj);
```

read_ups_testid : RFC1628의 upsTestId(가장 최근에 수행된 테스트)에 해당하는 값을 얻을 수 있습니다. 이 함수를 호출한 반환값이 TEST_USER_SPECIFIC이 아니면 upsTestId는 .1.3.6.1.2.1.33.1.7.7.RETURN_VALUE이 되며, usr_testindex는 무시됩니다. 만약 반환된 값이 TEST_USER_SPECIFIC이면 upsTestId는 read_usrtest(usr_testindex, usrtestobject)를 호출하여 얻어진 usrtestobject.oid가 됩니다.

```
UPS_TEST read_ups_testid(int *usr_testindex);
```

save_ups_testid : RFC1628의 upsTestId(가장 최근에 수행된 테스트)에 해당하는 값을 저장합니다. RFC1628을 준수하려면 UPS 테스트를 수행할 때 마다 반드시 이 함수를 호출하여 upsTestId를 저장해야 합니다.

```
int save_ups_testid(UPS_TEST testid, int usr_testindex);
```

RFC1628 준수 사항 : 이 함수를 호출하면 _UPS_DATA_TEST의 start_time이 자동으로 저장이 되어 테스트 종료시 _UPS_DATA_TEST의 elapse_time을 올바르게 계산할 수 있도록 해 줍니다. 만일 수행한 테스트가 사용자 지정 테스트라면 testid인자를 TEST_USER_SPECIFIC으로 하고 해당되는 usr_testindex를 올바르게 전달해 주어야 합니다. 수행한 테스트가 사용자 지정 테스트가 아니라면 usr_testindex에 그냥 0을 넘깁니다. testid가 TEST_ABORT 이면 _UPS_DATA_TEST의 elapse_time을 다시 계산하여 저장하고, _UPS_DATA_TEST의 summary를 TESTRES_ABORTED로 저장합니다. 그 외에는 _UPS_DATA_TEST의 summary를 TESTRES_INPROG로 저장합니다.

read_ups_test_summary, save_ups_test_summary : UPS 데이터 영역에 저장되어 있는 테스트 수행 결과 요약을 읽거나 씁니다.

```
UPS_TESTRES read_ups_test_summary();
int save_ups_test_summary(UPS_TESTRES summary);
```

read_ups_test_detail, save_ups_test_detail : UPS 데이터 영역에 저장되어 있는 테스트 수행 결과 상세 설명을 읽거나 씁니다.

```
int read_ups_test_detail(char *detail);
int save_ups_test_detail(char *detail);
```

get_ups_test_starttime : 마지막으로 테스트를 수행한 시각과 UPSLink가 동작을 시작한 시각과의 시간차(sysUpTime)를 100분의 1초 단위로 반환합니다. 수행된 테스트가 없을 경우에는 0을 반환합니다.

```
unsigned long get_ups_test_starttime();
```

get_ups_test_elapsetime : 마지막으로 수행한 또는 현재 수행중인 테스트의 소요시간을 100분의 1초 단위로 반환합니다. 수행된 테스트가 없을 경우에는 0을 반환합니다.

```
unsigned long get_ups_test_elapsetime();
```

ups_ser.h

ups_ser.h에는 UPS와 RS232통신과 관련된 함수들이 선언되어 있습니다.

ups_ser_open : 시스템의 UART설정에 따라 시리얼 포트를 엽니다. 프로그램 시작 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
int ups_ser_open();
```

ups_ser_close : 시리얼 포트를 닫습니다. 프로그램 종료 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
void ups_ser_close();
```

ups_ser_write : 시리얼 포트에 buff를 len만큼 씁니다. 실제로 써진 byte 수가 반환됩니다.

```
int ups_ser_write(const char *buff, int len);
```

ups_ser_read : 시리얼 포트에서 최대 len만큼 데이터를 읽어 buff에 저장합니다. 실제로 읽힌 byte 수가 반환됩니다.

```
int ups_ser_read(char *buff, int len);
```

ups_msgq.h

웹의 설정/제어/테스트 명령이나 SNMP-set 명령이 UPS 시리얼 프로그램에 전달되는 것과 관련된 함수들이 선언되어 있습니다. 또한 전달될 명령들이 enum으로 정의되어 있습니다.

init_upsmsgq : 명령 메시지 전달 구조를 초기화 합니다. 프로그램 시작 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
void init_upsmsgq(void);
```

send_upsmsgq : 명령 메시지를 보내는 함수 입니다. 이는 웹 서버나 SNMP 데몬에서 호출될 함수로 사용자는 절대 호출할 필요가 없습니다.

```
int send_upsmsgq(int command, long value, char *str);
```

read_upsmsgq : 전달된 명령 메시지가 있다면 그것을 읽는 함수입니다. ups_app.c의 템플릿 코드에 이미 구현이 되어 있습니다. 다만 명령을 받은 이후에 처리하는 handle_upsmsg(..) 함수를 사용자가 수정해야 합니다. command들은 ups_msgq.h에 enum으로 정의되어 있습니다. value와 str은 명령과 함께 전달된 숫자와 문자열 형태의 보조 데이터 입니다. 이것의 구체적인 사용법은 다음 장의 “ups_app.c 수정하기”를 참고하십시오.

```
int read_upsmsgq(int *command, long *value, char *str);
```

Note : 이 함수는 블로킹 함수 입니다.

ups_sem.h

시리얼 포트 액세스 시에 쓰레드 안전 보장을 위한 세마포어 유틸리티 입니다.

init_upssem : 세마포어를 초기화 합니다. 프로그램 시작 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
int init_upssem();
```

lock_upssem : 세마포어를 잠금니다. 시리얼 포트를 액세스 하기 직전에 호출합니다. 시리얼 포트 작업 이후 반드시 unlock_upssem을 호출하여 반환해야 합니다.

```
int lock_upssem();
```

unlock_upssem : 세마포어 잠김을 해제합니다. lock_upssem과 짝을 이루어 사용해야 합니다.

```
int unlock_upssem();
```

strqueue.h

시리얼 포트에서 읽은 문자열 데이터를 바로 사용하지 않고 버퍼에 저장한 후 사용할 수 있게 해주는 유틸리티 입니다. 문자열 형태의 데이터를 사용하기 때문에 시리얼 프로토콜이 바이너리 형태일 경우는 사용이 불가능하며, 아스키 형태일 경우에만 사용 가능합니다. 이것을 사용하려면 strqueue.h에 형태가 정의된 queue 변수 하나를 전역변수로 선언해 두어야 합니다. 기본적으로 ups_mon.c에 하나의 queue가 선언되어 있습니다. 바이너리 프로토콜을 사용할 시에는 삭제할 수 있습니다. 필요에 따라 다중의 버퍼도 사용 가능합니다.

```
typedef struct {
	char *buf;
	int maxsize;
} queue;
```

queue_init : 버퍼 메모리를 할당하고 초기화 합니다. 프로그램 시작 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다. 바이너리 프로토콜을 사용할 시에는 삭제할 수 있습니다. 최대 size는 65535 bytes입니다.

```
int queue_init(queue *q, int size);
```

queue_free : 버퍼 메모리를 할당을 해제하고 종결화 합니다. 프로그램 종료 시 ups_mon.c에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다. 바이너리 프로토콜을 사용할 시에는 삭제할 수 있습니다.

```
void queue_free(queue *q);
```

queue_maxsize : 버퍼에 할당된 메모리의 사이즈를 반환합니다.

```
int queue_maxsize(queue *q);
```

queue_size : 버퍼에 저장된 문자열의 길이를 반환합니다.

```
int queue_size(queue *q);
```

queue_write : buf가 가리키는 문자열을 최대 len만큼 버퍼의 뒤에 추가합니다.

```
int queue_write(queue *q, char *buf, int len);
```

queue_copy : 버퍼에 저장된 문자열을 q->buf+start 부터 len만큼 buf에 복사합니다. 버퍼의 내용은 그대로 남아있습니다.

```
int queue_copy(queue *q, char *buf, int start, int len);
```

queue_read : 버퍼에 저장된 문자열을 q->buf+start 부터 len만큼 buf에 잘라 붙입니다. 버퍼에서 잘린 이후의 문자열은 잘린 만큼 앞으로 당겨집니다.

```
int queue_read(queue *q, char *buf, int start, int len);
```

queue_delete : 버퍼에 저장된 문자열을 q->buf+start 부터 len만큼 삭제합니다. 버퍼에서 삭제된 이후의 문자열은 삭제된 만큼 앞으로 당겨집니다.

```
int queue_delete(queue *q, int start, int len);
```

queue_clear : 버퍼에 저장된 문자열을 모두 삭제합니다.

```
int queue_clear(queue *q);
```

ups_easytraplog_msgq.h

UPSLink는 libups_data.h에 있는 set_alarm(..) 함수를 사용하면 RF1628에 해당하는 SNMP 트랩을 자동으로 전송하게 되어 있으나 이와는 별개로 추가적인 SNMP 트랩을 전송하고자 할 때 유용하게 쓰일 수 있는 함수가 정의되어 있습니다.

Note : 설정에서 트랩타입이 EASY_TRAP으로 설정이 되어 있는 Trap receiver에게만 전송하며, 전송되는 트랩은 v1 입니다.

```
typedef enum {VB_INTEGER=1, VB_STRING=2, VB_OID=3} VB_TYPE;

struct _ups_easytrap_varbind{

      unsigned long oid[MAX_OID_DEPTH];
      int     int_var;
      char    string_var[200];
      unsigned long oid_var[             ];
};

struct _ups_easytraplogmsg {
      unsigned long enterprise_oid[MAX_OID_DEPTH];
      int     generic;
      int     specific;
      struct _ups_easytrap_varbind varbind[10];
      int             sendtrap; // 0-do not send, 1-send
      int             sendmail; // 0-do not send, 1-send
      int             writelog; // 0-do not write, 1-write
      char    desc[100];
};
```

전송할 트랩의 내용에 따라 위와 같이 정의된 구조체를 만들어 send_upseasytraplogmsgq (struct _ups_easytraplogmsg msg) 함수를 사용하여 트랩을 전송합니다. 최대 10개의 변수를 포함하여 트랩을 전송할 수 있습니다.

init_upseasytraplogmsgq : 트랩 전달 구조를 초기화 합니다. 프로그램 시작 시 ups_mon.c 에서 호출되므로 사용자가 별도로 호출할 필요가 없습니다.

```
void init_upseasytraplogmsgq(void);
```

send_upseasytraplogmsgq : 트랩을 전송하는 함수입니다.

```
int send_upseasytraplogmsgq(struct _ups_easytraplogmsg msg);
```

2.5. ups_app.c 수정하기

ups_app.c에서는 UPS와의 시리얼 통신, 프로토콜 해석 및 데이터 저장, 사용자의 설정 및 제어 명령을 UPS에 전달하는 기능을 수행합니다. 그 밖에 추가로 사용자 지정 항목의 추가가 가능합니다. 사용자 기능의 추가에 관해서는 4장에 자세히 설명되어 있습니다. 여기서는 ups_app.c를 개괄적으로 이해하고 기본적인 사항을 수정할 수 있도록 각 함수별로 구분하여 설명합니다.

> Note : ups_app.c는 사용자의 수정이 용이하도록 필수적으로 구현되어질 부분을 포함한 기본적인 구조가 갖추어진 형태로 제공됩니다.

start_ups_app : ups_mon.c에서 이 함수가 호출됩니다. 실질적인 UPS 시리얼 프로그램의 시작입니다. 여기에서는 프로세스를 2개로 나누어 각각 UPS 감시 기능과, 사용자 이벤트 처리를 담당하게 합니다. 아래 코드와 같이 있는 설명을 읽어보십시오.

```
void start_ups_app()
{
                      될 프로세스의 ID
   register_usrspec(); // 사용자 정의 항목 등록
   queue_clear(&upsq); // 버퍼 큐     용 삭제 (아스키 프로토콜 사용할 경우만 해당)
   pid = fork(); // 프로세스를 분
   if(pid == -1){ //        분리에             종료함
          fprintf(stderr, "p  ce s spawn fail. exiting..\r\n");
          exit(0);
   }
   else if(pid != 0){ // 부모 프로세스, UPS 감시를 담당
          // UPS 시리얼 프로그램의 버전 저장, 구조체 단위로 읽고 씀.
          _UPS_DATA_     ;

          strncpy(id.app_sw_ver, “user_app v1.0”, sizeof(id.app_sw_ver)-1);
          save_ups_identification(&id);

                                              의 시간 동기화를 위해 잠시 쉼 (0.5초)


          while(1){{
                           / 시리얼 포트 접근에 대한 쓰레드 안전을 위해 세마포어를 잠
금
                          lock_upssem();

                                                  기 메시지 처리… 등 일상적인 UPS
  시
                          // 내용이 poll_device 함수에 구현되어 있어야 합니다.
                          poll_device();
                          // poll_device


                          unlock_upssem();

                          // CPU를 양보하기 위해 잠시 쉼 (1초)
                          usleep(1000000);
                  }
          }
   }
   else{ /      프로세스, 사용자 이벤트를 처리
          while(1){
                  // 웹이
```

```
음.
                int command;
                long val;
                char strval[256] = {0, };

                // read_upsmsgq : 이 함수는 블로킹 함수입니다.
                // 메시지 큐에 메시지가 도착할 때 까지 진행하지 않습니다.
                read_upsmsgq(&command, &val, strval);

                // 메시지 큐에서 메시지가 읽힘

                // 시리얼 포트 접근에 대한 쓰레드 안전을 위해 세마포어를 잠금
                lock_upssem();

                // 사용자 명령을 처리함
                // 사용자 명령 구분자인 command는 ups_msgq.h에 그 종류가 정의됨
                // val과 strval은 각각 명령 처리에 필요한 숫자 또는 문자열 형태의
                // 인자값임
```

handle_upsmsg(command,

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#include "ups_ser.h"
#include "libups_data.h"
#include "ups_data.h"
#include "ups_msgq.h"
#include "strqueue.h"

// implementations specific includes..
// #include "USER_INCLUDE1.H"
// #include "USER_INCLUDE2.H"
// #include "USER_INCLUDE3.H"

#define UPS_TIME_INTERVAL 300000 // microsec
#define UPS_POLL_INTERVAL 1000000 // microsec
#define USER_APP_VER "VERSION" // the version of this application

extern queue upsq;
static int send_command(char *cmd);
static int read_buffer(int lastcmd);

/*
* handle_upsmsg(int command, long val, char *strval);
* handle any messages from the web or snmp set command.
* the messages are defined in ups_msgq.h
*/
int handle_upsmsg(int command, long val, char *strval)
{
    // send any set command to the UPS
    char cmdstr[100] = {0, };
    // int lastcmd;

    // TO DO :
    // conctruct the command
```

```
    // ex)
    // get_set_command(cmdstr, command, val, strval);

                    //
                    unlock_upssem();

                    // CPU              잠시 쉼 (1초)
                    usleep(1000000);
            }
    }
}
```

poll_device : UPS에 명령을 보내고 응답을 읽습니다. 여기서 호출된 read_buffer 함수에서 응답을 해석할 것입니다. 보내질 명령의 순서를 이 함수에서 판단하여 적용해야 합니다. 아래의 코드는 BATTERY부터 BYPASS까지 네 개의 명령을 순환하며 UPS를 감시하는 예제입니다.

```
enum { BATTERY=1, INPUT=2, OUTPUT=3, BYPASS=4};
int lastcmd = GET_BATTERY;

s
{
                            };


    {
                        :
                    sprintf(cmdstr, “GET_BATTERY”);
                    break;
            case INPUT:
                    sprintf(   str, “GET_INPUT”);
                    break;

                OU
                    sprintf(cmdstr, “GET_OUTPUT”);
                    break;

                BY
                    sprintf(cmdstr, “GET_BYPASS”);
                    break;
}// send constructed command
    send_command(cmdstr);
    usleep(500000); // 0.5초간 기다림
    read_buffer(lastcmd);


            lastcmd = BATTERY;
    e
            lastcmd
}
```

send_command : 시리얼 포트를 통해 UPS로 명령을 전송합니다. 수정이 불필요합니다.

```
static int send_command(char *cmd, int len)
{

    res = ups_ser_write(cmd, len);
    return res;
}
```

read_buffer : UPS의 응답을 시리얼 포트로부터 읽습니다. 응답을 문자열 버퍼 큐(upsq)에 저장한 후에 큐에 저장된 데이터를 해석합니다. (프로토콜이 바이너리 형태일 경우에는 문자열 버퍼 큐에 저장하는 과정 없이 바로 해석하거나 별도의 방법을 구현할 수도 있습니다.)

직전에 보낸 명령이 무엇이었는지 고려하여 해석한 내용을 UPS데이터에 저장합니다. 이 때 발생 또는 해제된 경보 내역도 set_alarm함수를 사용하여 설정 합니다. UPS로부터 읽힌 데이터에 명령에 대한 응답 이외에 비동기적인 경보 내역이 있다면 이것도 별도로 처리해 주어야 합니다.

Note : 함수의 인자로는 직전에 UPS로 전송된 명령의 종류를 알려 줍니다. 명령의 종류는 사용자가 정의해야 합니다.

아래의 코드는 시리얼로 데이터를 읽어서 비동기 경보를 처리하고, 응답을 해석해서 데이터를 저장하는 예를 보여줍니다. 여기서는 프로토콜의 형태가 아스키라고 가정합니다.

```
static int read_buffer(int lastcmd)
{
    int len;
    int totallen = 0;
    char buff[2048] = {0, };

    // 시리얼 포트 버퍼에 있는 데이터를 끝까지 읽어 문자열 큐에 저장합니다.
    do{
            len = ups_ser_read(buff, sizeof(buff));
            if(len > 0){
                    queue_write (&upsq, buff, len);
                    totallen += len;
            }
    }while(len > 0);

    // 명령에 대한 응답이 없다면 다시 한번 더 시도합니다.
    if(totallen == 0){
            usleep(500000); // 0.5초간 기다립니다.
            do{
                    len = ups_ser_read(buff, sizeof(buff));
                    if(len > 0)    {
                            queue_write (&upsq, buff, len);
                            totallen += len;
                    }
            }while(len > 0);

            // 읽혀진 데이터가 없다면 UPS데이터를 재설정한 후,
            // "통신 두절"  경보를 발생시킵니다.
            if(totallen == 0){
                    _UPS_DATA_ID id;
                    reset_upsdata();
                    // UPS 시리얼 프로그램의 버전은 재설정 시키지 않습니다.
                    read_ups_identification(&id);
                    strcpy(id.app_sw_ver, "user_app v1.0");
                    save_ups_identification(&id);
                    set_alarm(_ALARM_COMMLOST, ALARM_PRESENT, false);
                    return -1;
            }
    }
    // 명령에 대한 응답이 있다면 "통신 두절" 경보를 해제합니다.
    // 참고로 set_alarm을 이용한 경보의 발생/해제는 마지막 인자를 false로 줄 경우
    // 호출 회수와 상관없이UPSLink가 발생/해제 상태의 전환을 스스로 판단하므로
    // "통신 두절" 경보를 매번 해제해도 상관이 없습니다.
    set_alarm(_ALARM_COMMLOST, ALARM_RELEASED, false);
```

```
    // 비동기 경보내역이 있다면 처리합니다.
    catch_all_async_traps();

    // 최후의 명령에 대한 응답을 해석하여 UPS 데이터를 갱신합니다.
    catch_response(lastcmd);

    // 문자열 버퍼 큐의 내용을 모두 삭제합니다.
    queue_clear(&upsq);
    return 0;
}
```

catch_all_async_traps : 직전 명령에 대한 응답이 아닌 비동기적인 메시지가 UPS로부터 읽혔을 경우 이를 처리하는 루틴을 포함시킬 수도 있습니다.

```
    // TO DO :
    // confirm the command result
    // ex)
    // lastcmd = get_confirm_command(cmdstr, command, val, strval);
    // usleep(UPS_TIME_INTERVAL);
    // send_command(cmdstr);
    // usleep(UPS_TIME_INTERVAL);
    // read_buffer(lastcmd);
}

/*
* static int catch_all_async_traps();
* check any asynchronous messages from the ups and process it
*/
static int catch_all_async_traps()
{
    // 문자열 큐 upsq에 저장되어 있는 전체 응답에서 비동기 메시지가 있는지 검사해서
    // 해당되는 경우 경보를 보내거나 UPS데이터를 갱신하는 등 필요한 동작을 수행합니다.
    // 처리한 upsq의 내용은 queue_delete 함수를 이용해서 upsq에서 지워버려야 합니다.
}
```

catch_response : 마지막 명령에 대한 응답을 읽어서 처리합니다. 아래 코드는 직전 명령이 UPS의 출력 상태를 얻는 것이었고, 그에 대한 응답을 얻어서 처리하는 예제입니다.

UPS는 단상이며, 응답의 형태는 아스키로 아래와 같다고 가정합니다.

응답형태 : RESP_OUTPUT:SOURCE,VOLTAGE,FREQUENCY,CURRENT,POWER,LOAD:END

```
TO DO:
    // define user trap header and length
    #define USER_TRAP_HEADER "USER_ASYNC_MSG_HEADER"
    #define USER_TRAPLEN 10
    char *tp = NULL;

    // find out USER_TRAP_HEADER from the queue
    while((tp = strstr(upsq.buf, USER_TRAP_HEADER)))

            int start;
            int traplen = USER_TRAPLEN;
            char trapstr[1024] = {0, };
            char *ptmp;

            // async trap detected
            ptmp = tp;
            start = tp - upsq.buf;
            ptmp += strlen(USER_TRAP_HEADER);

            if(strlen(ptmp) >= traplen)
```

```
           {
                  if(traplen < sizeof(trapstr))
                  {
                         // read the async message from the queue
                         queue_read(&upsq, trapstr,
start+strlen(USER_TRAP_HEADER), traplen);
                         // delete header so that this trap be ignored at
next search
                         queue_delete(&upsq, start,
strlen(USER_TRAP_HEADER));

                         // TO DO:
                         // handle trap data.
                         // ex)
                         // set_alarm(_ALARM_USR1, ALARM_PRESENT, true);
                         continue;
                  }
           }
           // delete unprocessed garbage string from the queue
           queue_delete(&upsq, start, strlen(USER_TRAP_HEADER));
    }
}

/*
* static int catch_response(int lastcmd);
* check the response from the ups and process it.
*/
static int catch_response(int lastcmd)
{
    char *pstart = NULL;
    char *pend = NULL;
int start, end, len;// TO DO:
    // define user response header and length
    #define USER_RESP_HEADER "USER_RESP_HEADER"
    #define USER_RESPLEN 10

    char *tr = NULL;
    int start;
    int resplen = USER_RESPLEN;
    char respstr[1024] = {0, };

    switch(lastcmd)
    {
           .
           .
           .
           case OUTPUT:
                  문자열 버퍼에서 응답에 해당하는 부분의 시작과 끝을 찾음
                  pstart = strstr(upsq.buf, “RESP_OUTPUT”);
                  pend = strstr(upsq.buf, “END”);
                  if(pstart && pend)
                  {
                         char *p;
                         char *psource, *pvoltage, *pfrequency;
                         char *pcurrent, *ppower, *pload;
                         _UPS_DATA_OUTPUT output;

                         // 현재의 UPS 데이터를 읽음
                         read_ups_output(&output);

                         // 문자열 버퍼에서 응답에 해당하는 부분을 읽어냄
                         start = pstart - upsq.buf;
                         end = pend + strlen(END) - upsq.buf;
                         len = end-start;
                         queue_read(&upsq, respstr, start, len);

                         // 응답 해석
```

```
                        p = respstr + strlen("REST_OUTPUT:");
                        if(p) psource = strtok(p, ",");
                        if(psource) pvoltage = strtok(NULL, ",");
                        if(pvoltage) pfrequency = strtok(NULL, ",");
                        if(pfrequency) pcurrent = strtok(NULL, ",");
                        if(pcurrent) ppower = strtok(NULL, ",");
                        if(ppower) pload = strtok(NULL, " ");

                        // 해석한 데이터를 저장
                        output.num_lines = 1; // 단상으로 가정
                        if(psource) output.source = atoi(psource);
                        // ups_data.h에 명시된 frequency와 current의 단위에 주
의!
                        if(pfrequency) output.frequency = atoi(pfrequency)
* 10;
                        if(pvoltage) output.voltage[0] = atoi(pvoltage);
                        if(pcurrent) output.current[0] = atoi(pcurrent) *
10;
                        if(ppower) output.power[0] = atoi(ppower);
                        if(pload) output.load[0] = atoi(pload);

                        // 실질적으로 UPS 데이터 영역에 저장
                        save_ups_output(&output);

                        // 필요에 따라 경보 상태를 발생 또는 해제 상태로 설정합니
다.
                        if(output.source == OUTSRC_NORMAL){
                                set_alarm(_ALARM_ONBATT, ALARM_RELEASED,
true);
                                set_alarm(_ALARM_ONBYPASS, ALARM_RELEASED,
true);
                        }
                        else if(output.source == OUTSRC_BATTERY){
                                set_alarm(_ALARM_ONBATT, ALARM_PRESENT,
true);
                                set_alarm(_ALARM_ONBYPASS, ALARM_RELEASED,
true);
                        }
                        else if(output.source == OUTSRC_BYPASS){
                                set_alarm(_ALARM_ONBATT, ALARM_RELEASED,
true);
                                set_alarm(_ALARM_ONBYPASS, ALARM_PRESENT,
true);
                        }
                }
                break;
            .
            .
            .
    }
    return 0;
}
```

handle_upsmsg : 웹이나 SNMP-set 명령을 통해 사용자가 발생시킨 이벤트를 처리합니다. 인자로 넘겨진 command는 이벤트의 종류가 무엇인지 알려주며, 그 내용은 ups_msgq.h에 정의되어 있습니다. val과 strval은 이벤트와 관련된 숫자 또는 문자열 형태의 인자로 command에 따라서 숫자만 이용하거나 문자열만 또는 둘 다를 이용하기도 합니다. command에 따라서 UPS에 특정 명령을 전송하거나 그 외의 동작을 할 수도 있습니다. 아래 코드를 살펴보시기 바랍니다.

```
int handle_upsmsg(int command, long val, char *strval)
{
```

```
    // UPS에 전송할 명령 문자열 버퍼
cmdstr[100] = {0, };

    switch(command)
    {
            // 기본 정보
            case _UPSMSG_SET_ID_IDENTNAME: // UPS의 identname을 설정합니다.
                    sprintf(cmdstr, “SETIDNAME=%s”, strval); // 문자열 인자를 사
용
                    break;
            case _UPSMSG_SET_ID_ATTDEVICE:       // UPS에 부착된 장비를 설정합니
다.
                    sprintf(cmdstr, “SETATTDEV=%s”, strval); // 문자열 인자를 사
용
                    break;

            // 베터리
            case _UPSMSG_SET_BATT_REPLACEDATE:
                    // 배터리를 마지막으로 교체한 날짜, strval을 사용
                    break;

            // UPS 테스트
            case _UPSMSG_SET_TEST_ABORT:
                    // 테스트 취소, val, strval모두 사용하지 않습니다.
                    break;
            case _UPSMSG_SET_TEST_GENERAL:
                    // 일반 테스트 수행, val, strval모두 사용하지 않습니다.
                    break;
            case _UPSMSG_SET_TEST_QUICKBATT:
                    // 배터리 즉석 테스트 수행, val, strval모두 사용하지 않습니다.
                    break;
            case _UPSMSG_SET_TEST_DEEPBATT:
                    // 배터리 심층 테스트 수행, val, strval모두 사용하지 않습니다.
                    break;
            case _UPSMSG_SET_TEST_USR:
                    // 사용자 지정 테스트, 4장을 참고하십시오.
                    break;

            // UPS 제어
            case _UPSMSG_SET_CONT_SHUTDOWNTYPE:
                    // 셧다운 타입 설정, ups_data.h의 SHUTDOWN_TYPE형태의 val을 사
용
                    break;
            case _UPSMSG_SET_CONT_SHUTDOWNAFDELAY:
                    // 일정 시간 후에 UPS를 셧다운시킴, val은 초단위의 일정 시간을 나타
냄
                    break;
            case _UPSMSG_SET_CONT_STARTUPAFDELAY:
                    // 일정 시간 후에 UPS를 시작, val은 초단위의 일정 시간을 나타냄
                    break;
            case _UPSMSG_SET_CONT_REBOOTWDURATION:
                    // 일정 시간 후에 UPS를 재시작, val은 초단위의 일정 시간을 나타냄
                    break;
            case _UPSMSG_SET_CONT_AUTORESTARTTYPE:
                    // 셧다운 타입 설정, ups_data.h의 AUTO_RESTART형태의 val을 사용
                    break;

            // UPS 설정
            case _UPSMSG_SET_CONF_INPUTVOLT:
                    // 설정치 입력 전압을 설정, val은 volts단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_INPUTFREQ:
                    // 설정치 입력 주파수를 설정, val은 Hz단위의 설정값
                    break;
```

```
            case _UPSMSG_SET_CONF_OUTPUTVOLT:
                    // 설정치 출력 전압을 설정, val은 volts단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_OUTPUTFREQ:
                    // 설정치 출력 주파수를 설정, val은 Hz단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_OUTPUTVA:
                    // 설정치 출력 전압/전류 rating을 설정, val은 volt-amp단위의 설정
값
                    break;
            case _UPSMSG_SET_CONF_OUTPUTPOWER:
                    // 설정치 출력 파워를 설정, val은 watts단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_LOWBATTTIME:
                    // 설정치 low battery time을 설정, val은 분단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_AUDIBLESTATUS:
                    // 설정치 가청 경보 설정
                    // ups_data.h의 ADBLALM_STATUS형태의 val을 사용
                    break;
            case _UPSMSG_SET_CONF_LOWVOLTTRPT:
                    // 설정치 low battery transfer point 설정
                    // val은 volts단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_HIGHVOLTTRPT:
                    // 설정치 high battery transfer point 설정
                    // val은 volts단위의 설정값
                    break;
            case _UPSMSG_SET_CONF_BATTLIFE:
                    break;

            // 사용자 지정 제어 명령 - 4장을 참고하십시오.
            case _UPSMSG_SET_CONT_USR1:
            case _UPSMSG_SET_CONT_USR2:
            case _UPSMSG_SET_CONT_USR3:
            case _UPSMSG_SET_CONT_USR4:
            case _UPSMSG_SET_CONT_USR5:
            case _UPSMSG_SET_CONT_USR6:
            case _UPSMSG_SET_CONT_USR7:
            case _UPSMSG_SET_CONT_USR8:
            case _UPSMSG_SET_CONT_USR9:
            case _UPSMSG_SET_CONT_USR10:
                    break;

            // 사용자 지정 설정 명령 - 4장을 참고하십시오.
            case _UPSMSG_SET_CONF_USR1:
            case _UPSMSG_SET_CONF_USR2:
            case _UPSMSG_SET_CONF_USR3:
            case _UPSMSG_SET_CONF_USR4:
            case _UPSMSG_SET_CONF_USR5:
            case _UPSMSG_SET_CONF_USR6:
            case _UPSMSG_SET_CONF_USR7;
            case _UPSMSG_SET_CONF_USR8:
            case _UPSMSG_SET_CONF_USR9:
            case _UPSMSG_SET_CONF_USR10:
                    break;
            default:
                    break;
    }

    // 위에서 각 케이스별로 구성된 명령을 UPS에 전송하거나
    // 특정 동작을 수행합니다.
    // 위의 각 예에서 구성된 명령 문자열들은 사용자가 실제에 맞게 수정해야 할 부분입니다.
    char *ptmp;
```

```
    // find out USER_RESP_HEADER from the queue
    if(tr = strstr(upsq.buf, USER_RESP_HEADER))
    {
            // response detected
            ptmp = tr;
            start = tr - upsq.buf;
            ptmp += strlen(USER_RESP_HEADER);

            if(strlen(ptmp) >= resplen)
            {
                    if(resplen < sizeof(respstr))
                    {
                            // read the response from the queue
                            queue_read(&upsq, respstr,
start+strlen(USER_RESP_HEADER), resplen);
                            // delete header so that this response be ignored
at next search
                            queue_delete(&upsq, start,
strlen(USER_RESP_HEADER));
                            // TO DO:
                            // parse the response and process it.
                            // ex)
                            // parse_response(lastcmd, respstr);
                            return 0;
                    }
            }
            // delete unprocessed garbage string from the queue
            queue_delete(&upsq, start, strlen(USER_RESP_HEADER));
    }
}

/*
* static int read_buffer(int lastcmd);
* read the serial buffer and catch async messages and the response from the
ups
* also process the received messages.
*/
static int read_buffer(int lastcmd)
{
    int len;
    int totallen = 0;
    char buff[2048] = {0, };

    // read until there is no data remaining on the serial buffer
    do{
            // read from the serial
            len = ups_ser_read(buff, sizeof(buff));
            if(len > 0)
            {
                    // add read data to queue.
                    queue_write (&upsq, buff, len);
                    totallen += len;
            }
    }while(len > 0);

    // no data, try once more.
    if(totallen == 0)
    {
            // sleep for a wile
            usleep(500000);
            // retry once more
            do{
                    // read from the serial
                    len = ups_ser_read(buff, sizeof(buff));
                    if(len > 0)
                    {
                            // add read data to queue.
                            queue_write (&upsq, buff, len);
```

```
                        totallen += len;
                }
        }while(len > 0);

        if(totallen == 0)
        {
                // communication lost alarm
                _UPS_DATA_ID id;
                // reset ups data except app_sw_ver, alarm, .. see
libups_data.h
                reset_upsdata();
                read_ups_identification(&id);
                strcpy(id.app_sw_ver, USER_APP_VER);
                save_ups_identification(&id);
                // set communication lost alarm
                set_alarm(_ALARM_COMMLOST, ALARM_PRESENT, false);
                return -1;
        }
    }
    // release communication lost alarm since the communication was made
    set_alarm(_ALARM_COMMLOST, ALARM_RELEASED, false);

    // catch async trap if any
    catch_all_async_traps();
    // read the response and parse it
    catch_response(lastcmd);
    // clear queue (clear garbage or unknown)
    queue_clear(&upsq);
    return 0;
}

/*
* static int poll_device()
* poll the device. do this periodically.
*/
static int poll_device()
{
    char cmdstr[100] = {0, };
    int lastcmd;

    // TO DO:
    // construct the poll command
    // ex)
    // lastcmd = get_poll_command(cmdstr);
    send_command(cmdstr);
}
```

register_usrspec : 사용자 지정 데이터 항목을 추가하는 곳입니다. 4장을 참고해서 이 함수 안에 필요한 사용자 정보를 추가하면 됩니다.

```
    usleep(UPS_TIME_INTERVAL);
    read_buffer(lastcmd);
}

/*
* static int send_command(char *cmd);
* send command. write to serial buffer
*/
static int send_command(char *cmd)
{
    int res;
    res = ups_ser_write(cmd, strlen(cmd));
    return res;
}

/*
```

```
* static void register_usrspec();
* register user specific alarm and tests.
*/
static void register_usrspec()
{
    // 여기에 추가될 내용은 4장을 참고하시기 바랍니다.add user-specific alarm, info,
config, control and test here.

    // register user specific alarm
    {
            // the maximum length of the oid is defined in ups_data.h
MAX_OID_DEPTH 20 : max oid depth
            // user specific oid is 1.3.6.1.4.1.12236.1
            // ex)
            // #define USR_ALARM_OID_DEPTH1 8
            // unsigned long oid[USR_ALARM_OID_DEPTH1];
            // oid[0] = 1; oid[1] = 3; oid[2] = 6; oid[3] = 1; oid[4] = 4;
oid[5] = 1; oid[6] = 12236; oid[7] = 1;
            // register_usralarm(0, USR_ALARM_OID_DEPTH1, oid, "usr alarm 1",
"");
    }

    // register user specific infomation
    {
//          register_usrinfo(0, "user specific information 1",
INFO_TYPE_NUM);
//          save_ups_usrinfo(0, 100, NULL);
//          register_usrinfo(1, "user specific information 2",
INFO_TYPE_TEXT);
//          save_ups_usrinfo(1, 0, "abcde");
    }

    // register user specific configuration
    {
//          char *combo[] = {"item 1", "item 2", "item 3", "item 4", NULL};
//          register_usrconfig(3, "user specific configuration 1",
CONT_TYPE_COMBO, 2, NULL, combo);

//          register_usrconfig(0, "user specific configuration 2",
CONT_TYPE_NUM, 1234, NULL, NULL);
//          register_usrconfig(1, "user specific configuration 3",
CONT_TYPE_TEXT, 0, "default", NULL);
//          register_usrconfig(2, "user specific configuration 4",
CONT_TYPE_NONE, 0, NULL, NULL);
    }

    // register user specific control
    {
//          char *combo[] = {"Off", "On", NULL};
//          register_usrcontrol(0, "User specific control 1",
CONT_TYPE_COMBO, 1, NULL, combo);

//          register_usrcontrol(3, "user specific control 2", CONT_TYPE_NUM,
1234, NULL, NULL);
//          register_usrcontrol(4, "user specific control 3", CONT_TYPE_TEXT,
0, "default", NULL);
//          register_usrcontrol(5, "user specific control 4", CONT_TYPE_NONE,
0, NULL, NULL);
    }


    return;
}

/*
* void start_ups_app();
* main for this app. this will be called once initially by the ups_mon.
*/
```

```
void start_ups_app()
{
    pid_t pid;

    // register user-specific alarm and test
    register_usrspec();

    // clear queue initially
    queue_clear(&upsq);

    pid = fork();

    if(pid == -1)
    {
            fprintf(stderr, "process spawn fail. exiting..\r\n");
            exit(0);
    }
    else if(pid != 0)
    {
            // parent process

            // set the version of this application
            _UPS_DATA_ID id;
            read_ups_identification(&id);
            strcpy(id.app_sw_ver, USER_APP_VER);
            save_ups_identification(&id);

            usleep(500000);

            // poll ups
            while(1)
            {
                    {
                            // lock to be thead safe
                            lock_upssem();
                            // send scheduled command, read response and
asynchronous trap
                            poll_device();
                            // unlock
                            unlock_upssem();

                            // sleep for a while to yield the cpu.
                            usleep(UPS_POLL_INTERVAL);
                    }
            }
    }
    else
    {
            // child process
            // monitor ups messages from the web or snmp set command
            while(1)
            {
                    int command;
                    long val;
                    char strval[256] = {0, };

                    // this is a blocking function
                    read_upsmsgq(&command, &val, strval);

                    // lock to be thead safe
                    lock_upssem();
                    // received a command
                    // the handling may be write and read serial.
                    handle_upsmsg(command, val, strval);
                    // unlock
                    unlock_upssem();

                    // sleep for a while to yield the cpu.
```

```
                usleep(UPS_TIME_INTERVAL);
        }
    }
}
```

2.6. Makefile의 수정 및 컴파일

Note : 2.1절의 내용을 먼저 확인하시기 바랍니다.

ups_app.c를 사용자 프로토콜에 알맞게 수정을 하였고, 예를 들어 추가 사용자 코드인 user_code.c를 추가했다면 현재 작업 디렉터리에는 아래와 같이 되어 있습니다.

```
#ls [enter]
Makefile    include    lib    ups_app.c    ups_mon.c    user_code.c
```

링크할 C파일이 하나 더 추가되었으므로, Makefile을 수정합니다. 아래는 수정된 Makefile입니다. 굵게 표시되어 있는 부분이 추가된 부분입니다.

```
CROSS_COMPILE = /opt/hardhat/devkit/ppc/8xx/bin/ppc_8xx-
CC  = $(CROSS_COMPILE)gcc
AR  = $(CROSS_COMPILE)ar

INCLUDEDIRS = -I. -I./include
LDFLAGS = -L./lib
CFLAGS = ${INCLUDEDIRS} -D_REENTERANT

UPSAPP = upsapp
PROG_LIST = ${UPSAPP}

UPSAPP_OBJ = ./ups_mon.o ./ups_app.o ./user_code.o

all :  ${PROG_LIST}

${UPSAPP} : ${UPSAPP_OBJ}
      ${CC} -o $@ ${LDFLAGS} ${UPSAPP_OBJ} -lupslink

c.o :
      ${CC} -c ${CFLAGS} $<

clean :
      rm -f *.o ${UPSAPP_OBJ} ${PROG_LIST} core
```

컴파일한 후 결과물인 upsapp가 생성되었는지 확인합니다.

```
#make clean; make[enter]
#ls [enter]
Makefile     include       lib           ups_app.c      ups_app.o
ups_mon.c    ups_mon.o    user_code.c    user_code.o    upsapp
```

컴파일이 제대로 되었다면, 다음절을 참고하여 새로 작성된 UPS 시리얼 프로그램을 UPSLink에 다운로드 하여 실행시킵니다.

3. 다운로드 및 실행

3.1. 다운로드

제작된 사용자 UPS 시리얼 프로그램을 UPSLink의 웹 관리페이지에 접속하여 UPSLink로 다운로드 합니다.

Note : 파일의 사이즈는 128 Kbytes 이하여야 합니다. 텔넷이나 콘솔 메뉴에서는 사용자 UPS 시리얼 프로그램을 다운로드하는 것이 불가능합니다. 다운로드에 다소 시간이 걸릴 수 도 있습니다.

웹 메뉴 위치 : UPS management → UPS serial program

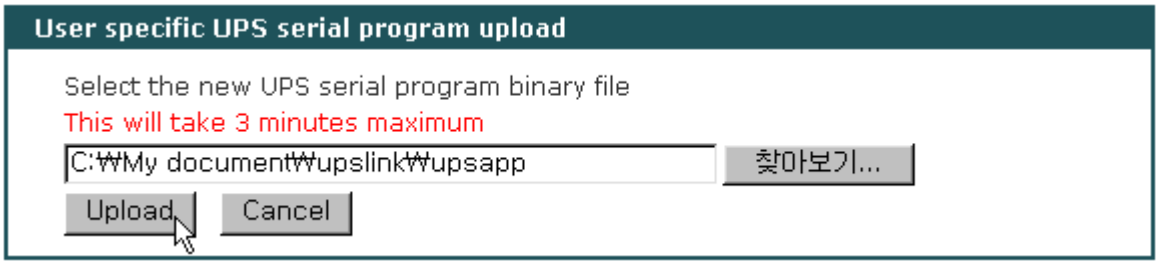

그림 3-1 사용자 UPS 시리얼 프로그램 다운로드

다운로드에 성공했다는 메시지를 확인합니다.

3.2. 사용자 UPS 시리얼 프로그램 실행

사용자 UPS 시리얼 프로그램의 다운로드에 성공했으면 새로운 UPS 시리얼 프로그램을 실행시킵니다. 메뉴의 UPS 시리얼 프로그램 선택에서 사용자 정의 프로그램을 선택한 후 '적용'버튼을 누릅니다.

웹 메뉴 위치 : UPS management → UPS serial program

그림 3-2 사용자 UPS 시리얼 프로그램 실행

새로운 UPS 시리얼 프로그램이 시작됩니다.

4. 사용자 지정 정보 추가하기

사용자 UPS 시리얼 프로그램에서는 프로토콜 수정과 더불어 RFC1628에 정의 되어 있지 않은 별도의 사용자 고유의 정보나 동작을 추가하는 것이 가능합니다.

추가 가능한 항목은 아래와 같습니다.

- **UPS 경보** : RFC1628에 명시 되어 있지 않으면서도 UPS와의 통신으로 감지될 수 있는 경보항목을 고유의 upsAlarmDescr와 함께 추가할 경우.
- **UPS 정보** : UPSLink에 저장되어 웹 화면에서 정적으로 고정된 값을 사용자에게 보여주려 할 경우, 예를 들어 UPS 관리자의 연락처.
- **UPS 상태 정보** : UPS와의 통신으로 얻을 수 있는 UPS상태 값. 예를 들어 현재 설치되어 있는 배터리의 개수.
- **UPS 설정** : UPS와의 통신으로 얻을 수 있는 UPS의 설정 값, 또는 역으로 UPS로 명령을 전송해 설정할 수 있는 값. 웹에서 설정 및 명령 전달을 위한 별도의 화면이 필요한 경우
- **UPS 제어** : 단지 UPS로 특정 명령을 전송해 UPS를 제어하고자 할 때, 그리고 전송을 위한 웹 화면이 필요한 경우.
- **UPS 테스트** : RFC1628에 명시 되어 있지 않은 별도의 사용자 테스트 항목을 고유의 upsTestId와 함께 추가할 경우.

4.1. 사용자 지정 UPS 경보

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 경보 내역을 추가하여 관리하고자 할 때 최대 5개의 사용자 지정 UPS 경보를 고유의 upsAlarmDescr와 함께 추가할 수 있습니다.

사용자 지정 경보의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usralarm(..) 함수를 이용하여 사용자 지정 경보를 등록합니다.

```
int register_usralarm(int usrindex, int oiddepth, unsigned long *oid,
char *shortdesc, char *longdesc);
```

usrindex : 등록될 사용자 지정 경보의 고유 인덱스 입니다. 0부터 4까지의 값이 가능합니다.
oiddepth : oid의 길이 입니다. 최대값은 20입니다.
oid : SNMP trap으로 전송될 경우 upsAlarmDescr에 해당될 OID 입니다.
shortdesc : 등록할 경보에 대한 요약 설명 입니다. 이 요약은 웹이나 텔넷등의 사용자 인터페이스에 해당 경보 발생시에 표시됩니다. (최대 31bytes)
longdesc : 등록할 경보에 대한 설명 입니다. (Null이 아닌 공백으로 처리 가능, 최대 255bytes)

read_usralarm(..)함수를 사용하여 기 등록된 사용자 지정 UPS 경보에 대한 정보를 얻을 수도 있습니다.

```
int read_usralarm(int usrindex, struct _usr_alarm_object *alarmobj);
```

예제

가령 2개의 사용자 지정 경보를 등록한다면 등록할 경보들의 oid가 각각 “1.3.6.1.4.1.12236.1” 과 “1.3.6.1.4.1.12236.2”이고 이 경보들의 shortdesc가 각각 “usr alarm 1”과 “usr alarm 2” 이라면, 아래 내용을 register_usrspec() 함수 안에 추가합니다.

```
unsigned long oid1[8];
unsigned long oid2[8];
oid1[0] = 1; oid1[1] = 3; oid1[2] = 6; oid1[3] = 1; oid1[4] = 4; oid1[5] =
1; oid1[6] = 12236; oid1[7] = 1;
oid2[0] = 1; oid2[1] = 3; oid2[2] = 6; oid2[3] = 1; oid2[4] = 4; oid2[5] =
1; oid2[6] = 12236; oid2[7] = 2;
// 위에서 정의된 두 개의 oid는 사용자가 실제에 맞게 수정해야 합니다.
register_usralarm(0, 8, oid1, "usr alarm 1", "");
register_usralarm(1, 8, oid2, "usr alarm 2", "");
```

usrindex = 0 으로 등록한 경보가 발생하였을 경우,

```
set_alarm(_ALARM_USR1, ALARM_PRESENT, true);
```

usrindex = 1 으로 등록한 경보가 해제되었을 경우,

```
set_alarm(_ALARM_USR2, ALARM_RELEASED, true);
```

을 각각 ups_app.c의 특정 위치에서 호출하여 해당 경보의 발생/해제 내역을 UPS 데이터 영역에 저장합니다.

Note : 사용자 경보에 등록된 usrindex(0~4)값에 따라 _ALARM_USR1 부터 _ALARM_USR5 까지를 지정해 줍니다.

Note : set_alarm()함수의 자세한 사용법은 3장을 참고하시기 바랍니다.

시스템 설정 상태에 따라서 자동으로 경보 로그 기록, SNMP트랩 전송 그리고 경보 이메일 전송이 이루어 집니다. 사용자가 등록한 경보가 발생 및 저장되었을 경우 해당 상태를 웹과 텔넷을 통해 확인할 수 있습니다.

아래 그림 4-1은 위에서 등록한 경보 중 첫번째 경보가 발생 및 저장되었을 경우 웹 화면을 보여줍니다.

웹 메뉴 위치 : UPS management → UPS status monitor

그림 4-1 사용자 경보 발생

4.2. 사용자 지정 UPS 정보

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 UPS 관련 정보를 추가하여 관리하고자 할 때 최대 10개의 사용자 지정 UPS 정보를 추가할 수 있습니다.

사용자 지정 경보의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usrinfo(..) 함수를 이용하여 사용자 지정 정보를 등록합니다.

```
int register_usrinfo(int usrindex, char *desc, INFO_TYPE type);
```

usrindex : 등록될 사용자 지정 정보의 고유 인덱스 입니다. 0부터 9까지의 값이 가능합니다.
desc : 등록될 정보에 대한 설명입니다. (최대 127bytes)
type : 등록될 정보의 형태입니다. 숫자 또는 문자열 중에 선택 가능합니다. (ups_data.h의 INFO_TYPE에 정의되어 있습니다.)

read_ups_usrinfo(..)함수와 save_ups_usrinfo(..)를 사용하여 기 등록된 사용자 지정 UPS 정보를 읽거나 동적으로 변경할 수 있습니다.

```
int read_ups_usrinfo(int usrindex, int *infoval, char *infostr);
int save_ups_usrinfo(int usrindex, int infoval, char *infostr);
```

save_ups_usrinfo 에서 infoval은 숫자 정보, infostr은 문자열 정보입니다. 사용자 지정 UPS 정보 등록 당시의 형태에 따라 해당 값을 넘기면 됩니다. 아래 예제를 참고하십시오. (infostr은 최대 63bytes 입니다.)

예제

UPSLink 웹 관리 페이지에서 UPS의 사이즈 폭과 UPS의 관리자 연락처를 보여주고 싶다면 아래 내용을 register_usrspec() 함수 안에 추가합니다.

```
register_usrinfo(0, "UPS physical dimension width [inch]", INFO_TYPE_NUM);
register_usrinfo(1, "UPS administrator's contact address", INFO_TYPE_TEXT);
```

위에서 등록된 2개의 정보는 기본적으로 각각 0과, 공백문자열을 사용자 정보 데이터에 저장하고 있습니다.

위에서 등록한 정보의 값을 변경하려면

```
save_ups_usrinfo(0, 45, NULL);
save_ups_usrinfo(1, 0, "Utility room 104 (tel. 1234-5678)");
```

와 같이 save_ups_usrinfo(..) 함수를 사용하여 그 값을 변경하여 저장해 줍니다.

아래 그림 4-2는 위에서 등록하고 변경한 정보가 웹 화면에 표시된 모습을 보여줍니다.

웹 메뉴 위치 : UPS management → UPS information

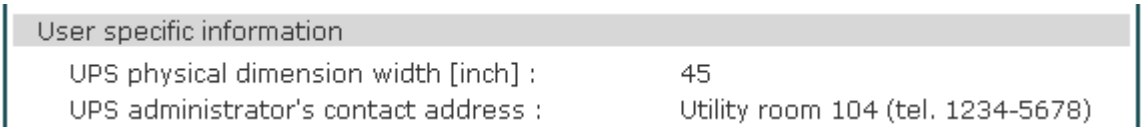

그림 4-2 사용자 정보 표시 화면

4.3. 사용자 지정 UPS 상태 정보

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 UPS 상태 정보를 추가하여 감시하고자 할 때 최대 10개의 사용자 지정 UPS 상태 정보를 추가할 수 있습니다.

사용자 지정 UPS 상태 정보의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usrstatus(..) 함수를 이용하여 사용자 지정 UPS 상태 정보를 등록합니다.

```
int register_usrstatus(int usrindex, char *desc, STATUS_TYPE type);
```

usrindex : 등록될 사용자 지정 UPS 상태 정보의 고유 인덱스 입니다. 0부터 9까지의 값이 가능합니다.
desc : 등록될 UPS 상태 정보에 대한 설명입니다. (최대 127bytes)
type : 등록될 UPS 상태 정보의 형태입니다. 숫자 또는 문자열 중에 선택 가능합니다. (ups_data.h의 STATUS_TYPE에 정의되어 있습니다.)

read_ups_usrstatus(..)함수와 save_ups_usrstatus(..)를 사용하여 등록된 사용자 지정 UPS 상태 정보를 읽거나 동적으로 변경할 수 있습니다.

```
int read_ups_usrstatus(int usrindex, int *statusval, char *statusstr);
int save_ups_usrstatus (int usrindex, int statusval, char *statusstr);
```

save_ups_usrstatus 에서 statusval은 숫자 정보, statusstr은 문자열 정보입니다. 사용자 지정 UPS 상태 정보 등록 당시의 형태에 따라 해당 값을 넘기면 됩니다. 아래 예제를 참고하십시오. (statusstr은 최대 63bytes 입니다.)

예제

UPSLink 웹 관리 페이지에서 UPS에 부착된 배터리의 개수와 UPS가 마지막으로 켜진 시각 보여주고 싶다면 아래 내용을 register_usrspec() 함수 안에 추가합니다.

```
register_usrstatus(0, "Number of battery cells", STATUS_TYPE_NUM);
register_usrstatus(1, "Last boot up data and time", STATUS_TYPE_TEXT);
```

위에서 등록된 2개의 정보는 기본적으로 각각 0과, 공백문자열을 사용자 지정 UPS 상태 정보 데이터에 저장하고 있습니다.

Note : 위 예제에서는 UPS와의 시리얼 통신으로 배터리 개수와 마지막 켜진 시각을 얻을 수 있다고 전제합니다.

사용자 프로토콜을 사용한 UPS와의 통신에서 배터리 개수는 20개, UPS의 시리얼 넘버는 “MYUPS-0176583”이라는 정보를 얻었다면

```
save_ups_usrstatus(0, 20, NULL);
save_ups_usrstatus(1, 0, "MYUPS-0176583");
```

와 같이 ups_app.c의 특정 부분에 save_ups_usrstatus(..) 함수를 사용하여 그 값을 변경하여 저장해 줍니다.

아래 그림 4-3는 위에서 등록하고 변경한 정보가 웹 화면에 표시된 모습을 보여줍니다.

웹 메뉴 위치 : UPS management → UPS status monitor

그림 4-3 사용자 UPS 상태 정보 표시 화면

4.4. 사용자 지정 UPS 설정

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 UPS 설정 항목을 최대 10개 까지 추가할 수 있습니다

사용자 지정 UPS 설정을 등록하게 되면 UPSLink의 관리 웹 페이지에 등록된 설정 명령을 내릴 수 있는 새로운 사용자 인터페이스가 만들어 집니다. 이 인터페이스를 사용하여 설정 값을 변경한 후 '적용'버튼을 누르면, 설정 값이 UPSLink에 저장이 되고 동시에 시리얼 포트를 통해 UPS에 설정 명령이 전송됩니다. 여기에서 저장과 명령 전송이 이루어 지는 과정은 ups_app.c를 사용자가 수정해야 합니다.

Note : 설정 값은 UPSLink의 휘발성 메모리 영역에 저장이 되므로 전원을 끄게 되면 그 값이 저장되지 않습니다.

아래 그림을 참고하십시오.

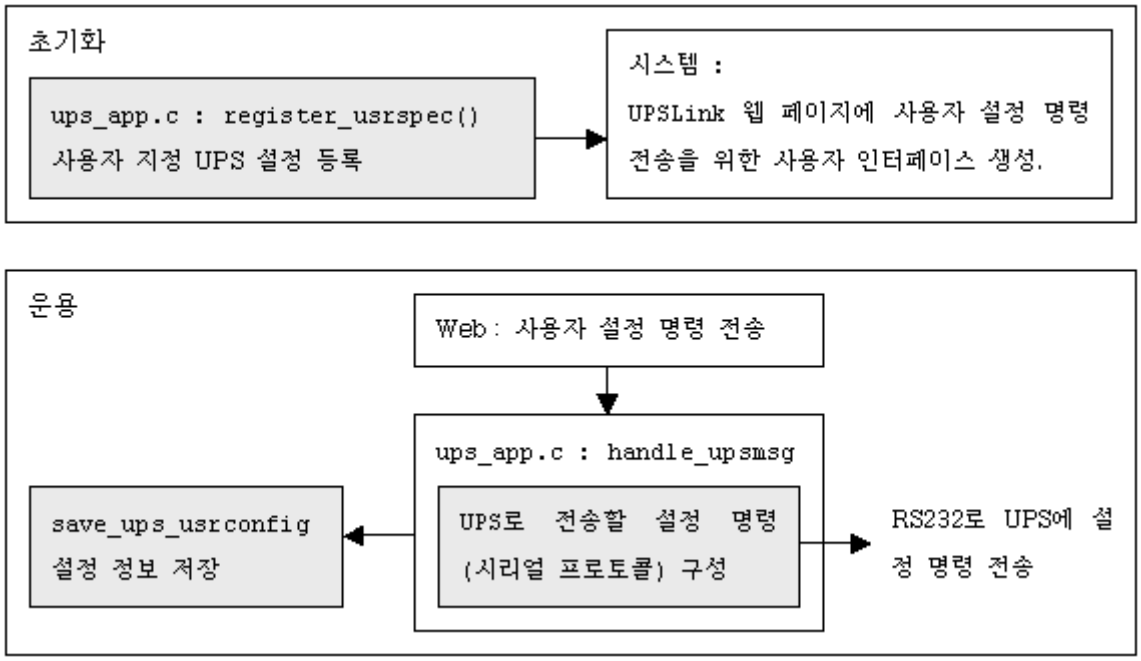

그림 4-4 사용자 지정 UPS 설정

그림 4-4에서 음영 표시된 부분은 사용자가 수정하거나 추가해야 할 부분입니다.

사용자 지정 UPS 설정의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usrconfig(..) 함수를 이용하여 사용자 지정 UPS 설정을 등록합니다.

```
int register_usrconfig(int usrindex, char *desc, CONT_TYPE type, char
**pldn_desc);
```

usrindex : 등록될 사용자 지정 UPS 설정의 고유 인덱스 입니다. 0부터 9까지의 값이 가능합니다.
desc : 등록될 UPS 설정에 대한 설명입니다. (최대 127bytes)
type : 등록될 UPS 설정의 형태입니다. 숫자, 문자열 또는 풀다운 메뉴 중에 선택 가능합니다. (ups_data.h의 CONT_TYPE에 정의되어 있습니다.)

pldn_desc : 등록될 UPS 설정이 풀다운 메뉴일 경우 메뉴 항목을 입력해 줍니다. (최대 31bytes 길이의 메뉴 항목을 10개까지 등록 가능합니다.)

read_ups_usrconfig(..)함수와 save_ups_usrconfig(..)를 사용하여 등록된 사용자 지정 UPS 설정값을 읽거나 동적으로 변경할 수 있습니다.

```
int read_ups_usrconfig(int usrindex, int *configval, char *configstr);
int save_ups_usrconfig(int usrindex, int configval, char *configstr);
```

save_ups_usrconfig 에서 configval은 숫자 설정 값 또는 풀다운 메뉴의 설정 인덱스, configstr은 문자열 설정 값입니다. 사용자 지정 UPS 설정 등록 당시의 형태에 따라 해당 값을 넘기면 됩니다. 아래 예제를 참고하십시오. (configstr은 최대 63bytes 입니다.)

예제

UPSLink 웹 관리 페이지에서 UPS의 최대 출력 전압, UPS에 부착된 장비의 고유 ID, 그리고 UPS의 시리얼 포트의 baudrate을 설정하고 싶다면 아래 내용을 register_usrspec() 함수 안에 추가합니다.

```
char *pldn[] = {"1200", "2400", "4800", "9600", "19200", "38400",
    "57600", "115200", "230400", NULL};
register_usrconfig(0, "Max. output voltage [volt]", CONT_TYPE_NUM,  NULL);
register_usrconfig(1, "ID of attached device", CONT_TYPE_TEXT, NULL);
register_usrconfig(2, "Baudrate of the UPS [bps]", CONT_TYPE_PLDN, pldn);
```

등록된 type이 CONT_TYPE_PLDN (풀다운 메뉴) 일 때만 pldn_desc를 넘겨줍니다. pldn_desc은 끝에 항상 NULL을 포함해야 합니다. (cy: ?)

Note : 위 예제에서는 UPS와의 시리얼 통신으로 각 항목의 설정 값을 읽거나 설정 명령을 전송할 수 있다고 전제합니다

위에서와 같이 사용자 UPS 설정을 등록해 두면 UPSLink 관리 웹에는 아래의 그림 4-5와 같은 화면이 생성됩니다.

웹 메뉴 위치 : UPS management → UPS configuration

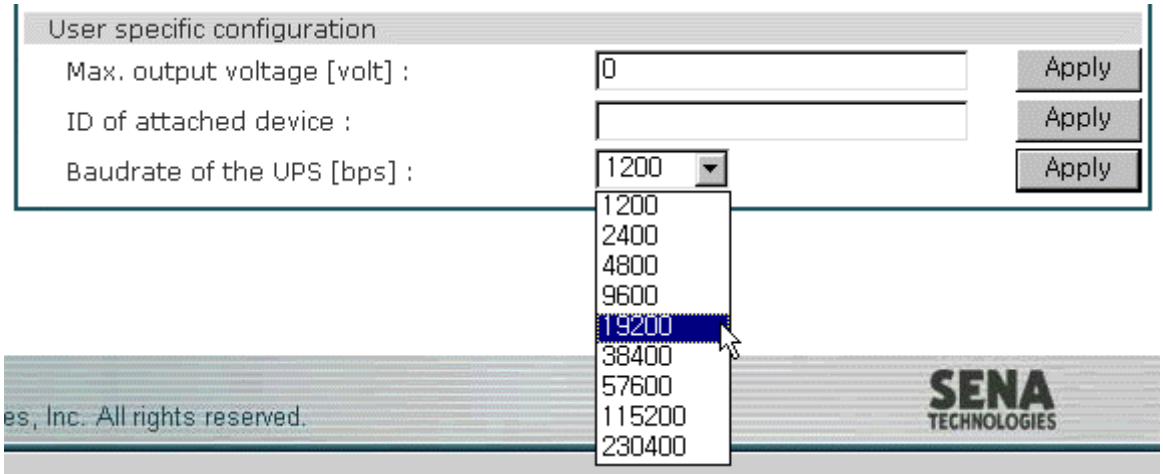

그림 4-5 등록된 사용자 지정 UPS 설정

UPS와 통신을 하고 있고, 현재 UPS에 설정되어 있는 값을 각각 340, ATTID 그리고 9600이라고 읽었다면, 아래와 같은 내용을 ups_app.c의 알맞은 위치에 추가해야 합니다.

```
save_ups_usrconfig(0, 340, NULL);
```

```
save_ups_usrconfig(1, 0, “ATTID”);
save_ups_usrconfig(2, 3, NULL);
```

위와 같이 UPS 설정을 저장해 주었을 경우 웹 화면은 그림 4-6과 같이 보여집니다.

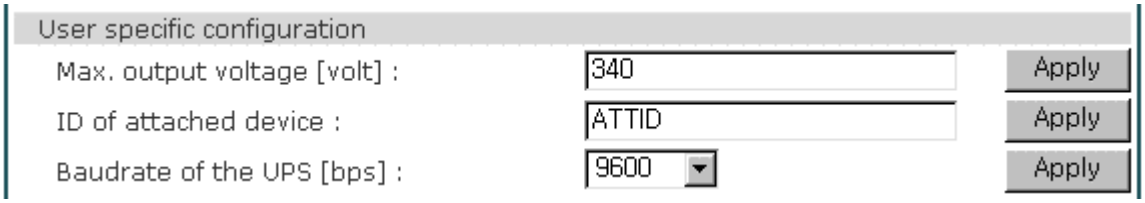

그림 4-6 UPS에서 현재 설정 값을 읽어 저장한 상태

그림 4-5에서 원하는 설정을 입력한 후 ‘적용’ 버튼을 눌렀을 경우 UPSLink의 내부 메시지 전달 메커니즘에 의해 ups_app.c의 handle_upsmsg(..)가 호출 됩니다. 사용자가 적용한 것을 시리얼 포트를 통해 UPS에 전송하기 위해서는 handle_upsmsg(..) 함수를 수정하여야 합니다. 아래의 예를 참고하십시오.

```
int handle_upsmsg(int command, long val, char *strval)
{
    char cmdstr[100] = {0, };
    switch(command)
    {
                    .
                    .
                    .
            case UPSMSG_SET_CONF_USR1:
                    sprintf(cmdstr, “SETMAXVOLT=%d”, (int)val);
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
            case UPSMSG_SET_CONF_USR2:
                    sprintf(cmdstr, “SETATTID=%s”, strval);
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
            case UPSMSG_SET_CONF_USR3:
                    {
                            char baudrate[10] = {0, };
                            switch(val)
                            {
                                    case 0: strcpy(baudrate, "1200"); break;
                                    case 1: strcpy(baudrate, "2400"); break;
                                    case 2: strcpy(baudrate, "4800"); break;
                                    case 3: strcpy(baudrate, "9600"); break;
                                    case 4: strcpy(baudrate, "19200"); break;
                                    case 5: strcpy(baudrate, "38400"); break;
                                    case 6: strcpy(baudrate, "57600"); break;
                                    case 7: strcpy(baudrate, "115200"); break;
                                    case 8: strcpy(baudrate, "230400"); break;
                            }
                            sprintf(cmdstr, "SETBR=%s", baudrate);
                    }
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
    }
    send_command(cmdstr); //UPS로 위에서 구성된 명령을 전송합니다.
}
```

Note : handle_upsmsg(..)에 대한 자세한 설명은 3절을 참고하시기 바랍니다

등록 시에 type이 문자열이었다면 strval을 설정 값으로 받아서 사용해야 하며, type이 숫자 또는 풀다운 메뉴였을 경우는 val을 설정값으로 받아서 사용해야 합니다. 다만 풀다운 메뉴에서

val은 실제 값이 아닌 사용자가 선택한 index입니다.

4.5. 사용자 지정 UPS 제어

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 UPS 제어 항목을 최대 10개 까지 추가할 수 있습니다.

사용자 지정 UPS 제어를 등록하게 되면 UPSLink의 관리 웹 페이지에 등록된 제어 명령을 내릴 수 있는 새로운 사용자 인터페이스가 만들어 집니다. 이 인터페이스를 사용하여 원하는 제어 명령에서 '수행'버튼을 누르면 시리얼 포트를 통해 UPS에 제어 명령이 전송됩니다. 여기에서 명령 전송이 이루어 지는 과정은 ups_app.c를 사용자가 수정해야 합니다.

사용자 지정 제어와 관련된 모든 항목은 앞 절의 사용자 지정 UPS 설정과 유사합니다. 다만 앞 절의 제어에서는 설정 값이 UPSLink에 저장되었던 것에 반해 제어 항목에서는 아무런 정보도 저장하지 않고 UPS에 명령만 전달하고 맙니다.

아래 그림을 참고하십시오.

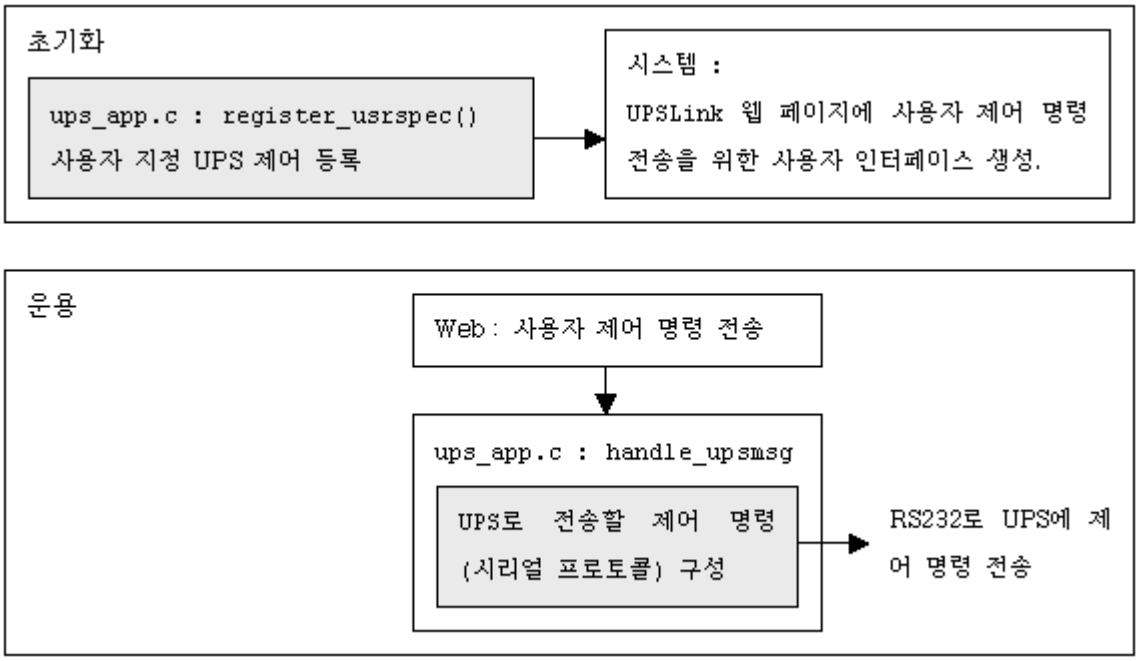

그림 4-7 사용자 지정 UPS 제어

그림 4-7에서 음영 표시된 부분은 사용자가 수정하거나 추가해야 할 부분입니다.

사용자 지정 UPS 제어의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usrcontrol(..) 함수를 이용하여 사용자 지정 UPS 제어를 등록합니다.

```
int register_usrcontrol(int usrindex, char *desc, CONT_TYPE type, int
defval, char *defstr, char **pldn_desc);
```

usrindex : 등록될 사용자 지정 UPS 제어의 고유 인덱스 입니다. 0부터 9까지의 값이 가능합니다.
desc : 등록될 UPS 제어에 대한 설명입니다. (최대 127bytes)
type : 등록될 UPS 제어의 형태입니다. 숫자, 문자열 또는 풀다운 메뉴 중에 선택 가능합니다. (ups_data.h의 CONT_TYPE에 정의되어 있습니다.)
defval : 등록될 UPS 제어의 형태가 숫자 또는 풀다운 메뉴일 경우 웹에 기본적으로 표시될 값 또는 기본 인덱스 입니다.
defstr : 등록될 UPS 제어의 형태가 문자열일 경우 웹에 기본적으로 표시될 문자열입니다.
pldn_desc : 등록될 UPS 제어가 풀다운 메뉴일 경우 메뉴 항목을 입력해 줍니다. (최대 31bytes 길이의 메뉴 항목을 10개까지 등록 가능합니다.)

제어 항목은 UPSLink에 아무런 데이터도 저장하지 않으므로 앞의 설정에서와 같이 save 또는 read 함수를 제공하지 않습니다. 웹에서도 항상 defval 또는 defstr 등으로 지정된 값을 표시합니다.

예제

사용자 지정 UPS 제어의 등록은 앞 절의 설정을 등록하는 방법과 사용하는 함수가 다른 것을 제외하고 같습니다. 또한 ups_app.c를 수정하는 부분 역시 등록한 후 handle_upsmsg(..)를 수정했던 방법에서 UPSMSG_SET_CONF_USR1~10 대신 UPSMSG_SET_CONT_USR1~10을 사용하는 것을 제외하고 동일합니다.

Note : 4.5절의 예제를 참고하시기 바랍니다.

4.6. 사용자 지정 UPS 테스트

네트워크를 통한 UPS의 관리를 위한 명세인 RFC1628에 정의 되어 있지 않은 테스트 항목을 추가하여 관리하고자 할 때 최대 5개의 사용자 지정 UPS 테스트를 고유의 upsTestId와 함께 추가할 수 있습니다.

사용자 지정 UPS 테스트의 등록

ups_app.c의 register_usrspec() 함수 내에 아래의 register_usrtest(..) 함수를 이용하여 사용자 지정 UPS 테스트를 등록합니다.

```
int register_usrtest(int usrindex, int oiddepth, unsigned long *oid,
char *testdesc);
```

usrindex : 등록될 사용자 지정 테스트의 고유 인덱스 입니다. 0부터 4까지의 값이 가능합니다.
oiddepth : oid의 길이 입니다. 최대값은 20입니다.
oid : upsTestId의 SNMP-get 질의에 대한 응답이 될 OID 입니다.
testdesc : 등록할 테스트에 대한 요약 설명 입니다. (최대 63bytes)

예제

UPSLink 웹 관리 페이지의 UPS 테스트 항목에 UPS의 정류기 테스트 항목과 인버터 테스트 항목을 을 추가하고 싶다면 아래 내용을 register_usrspec() 함수 안에 추가합니다.

```
unsigned long oid1[8];
unsigned long oid2[8];
oid1[0] = 1; oid1[1] = 3; oid1[2] = 6; oid1[3] = 1; oid1[4] = 4; oid1[5] =
1; oid1[6] = 12236; oid1[7] = 1;
```

```
oid2[0] = 1; oid2[1] = 3; oid2[2] = 6; oid2[3] = 1; oid2[4] = 4; oid2[5] =
1; oid2[6] = 12236; oid2[7] = 2;
// 위에서 정의된 두 개의 oid는 사용자가 실제에 맞게 수정해야 합니다.
register_usrtest(0, 8, oid1, “Rectifier test”);
register_usrtest(1, 8, oid2, “Inverter test”);
```

Note : 위와 같이 등록하고 UPSLink 웹에서 정류기 테스트 수행 명령을 내린 후, upsTestId를 SNMP-Get으로 질의하면 마지막으로 수행된 테스트의 OID인 .1.3.6.1.4.1.12236.1을 응답으로 얻게 됩니다.

Note : 위 예제에서는 UPS와의 시리얼 통신으로 각각의 테스트 수행 명령을 전송할 수 있다고 전제합니다

위에서와 같이 사용자 UPS 테스트를 등록해 두면 UPSLink 관리 웹에는 아래의 그림 4-8와 같이 UPS 테스트 풀다운 메뉴에 새로 추가한 테스트가 추가됩니다.

웹 메뉴 위치 : UPS management → UPS control

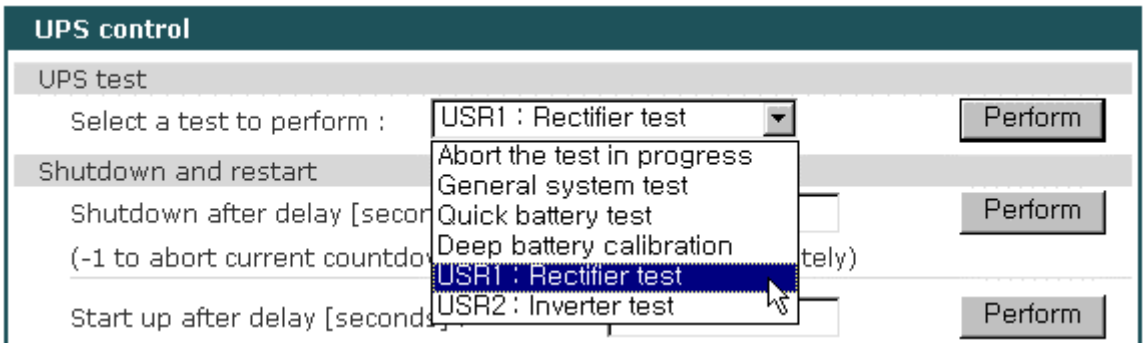

그림 4-8 등록된 사용자 지정 UPS 테스트

그림 4-8에서 추가된 테스트를 선택한 후 ‘수행’ 버튼을 눌렀을 경우 UPSLink의 내부 메시지 전달 메커니즘에 의해 ups_app.c의 handle_upsmsg(..)가 호출 됩니다. 사용자가 수행시킨 테스트 명령을 시리얼 포트를 통해 UPS에 전송하기 위해서는 handle_upsmsg(..) 함수를 수정하여야 합니다. 아래의 예를 참고하십시오.

```
int handle_upsmsg(int command, long val, char *strval)
{
    char cmdstr[100] = {0, };
    switch(command)
    {
                    .
                    .
                    .

            case _UPSMSG_SET_TEST_ABORT:
                    save_ups_testid(TEST_ABORT, 0);
                    sprintf(cmdstr, "TESTABORT");
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
            case _UPSMSG_SET_TEST_GENERAL:
                    save_ups_testid(TEST_GENERAL, 0);
                    sprintf(cmdstr, "GENERALTEST");
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
            case _UPSMSG_SET_TEST_QUICKBATT:
                    save_ups_testid(TEST_QUICKBATT, 0);
                    sprintf(cmdstr, "QUICKBATTTEST");
                    // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                    break;
            case _UPSMSG_SET_TEST_DEEPBATT:
```

```
                save_ups_testid(TEST_DEEPBATTCALIB, 0);
                sprintf(cmdstr, "DEEPBATTTEST");
                // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                break;
        case _UPSMSG_SET_TEST_USR:
                switch(val)
                {
                        case 0:
                                save_ups_testid(TEST_USER_SPECIFIC, 0);
                                sprintf(cmdstr, "RECTIFIERTEST");
                                break;
                        case 1:
                                save_ups_testid(TEST_USER_SPECIFIC, 1);
                                sprintf(cmdstr, "INVERTERTEST");
                                break;
                }
                // 위에서 구성되는 명령어는 사용자가 실제에 맞게 재 구성해야 합니다.
                break;
   }
   send_command(cmdstr); //UPS로 위에서 구성된 테스트 명령을 전송합니다.
}
```

위에서 모든 사용자 지정 테스트는 command 값이 _UPSMSG_SET_TEST_USR로 전달됩니다. 사용자 지정 테스트의 구별은 또 다른 인자인 val을 이용합니다. val은 테스트가 등록된 인덱스입니다.

Note : 명령 전송과 동시에 save_ups_testid(..)함수를 호출하여 upsTestId와 테스트 시작시각을 자동으로 저장하게 하는 것을 잊지 마시기 바랍니다.

Note : handle_upsmsg(..)과 save_ups_testid(..) 에 대한 자세한 설명은 3절을 참고하시기 바랍니다