Oprogramowanie testera wielokanałowych zasilaczy wysokiego napięcia dla detektorów krzemowych eksperymentu ATLAS w CERNie

Praca magisterska

Szymon Łukasik

Promotor: prof. dr hab. inż. Piotr Malecki

Kraków, czerwiec 2005

„An expert is a man who has made all the mistakes which can be made in a very narrow field"

Niels Bohr

„Computer science is no more about computers than astronomy is about telescopes"

E.W. Dijkstra

Autor składa serdeczne podziękowania

prof. Piotrowi Maleckiemu za inspirację i wsparcie,

inż. Edwardowi Górnickiemu i mgr Stefanowi Kopernemu za okazaną życzliwość i wszelką pomoc przy tworzeniu tej pracy,

oraz przyjacielowi Zenkowi Cygankowi...

za to że jest.

***

Pracę tą Rodzicom dedykuję

Spis treści

Wstęp

Celem niniejszej pracy było zaprojektowanie, wykonanie i uruchomienie oprogramowania urządzenia testującego wielokanałowe zasilacze wysokiego napięcia pracujące w ramach eksperymentu ATLAS w CERNie (tester ten wykonany został w ramach odrębnej pracy dyplomowej). Tytułem wprowadzenia warto poświęcić trochę miejsca samemu eksperymentowi oraz roli jaką pełnią w nim zasilacze i układ testujący dlań wykonany.

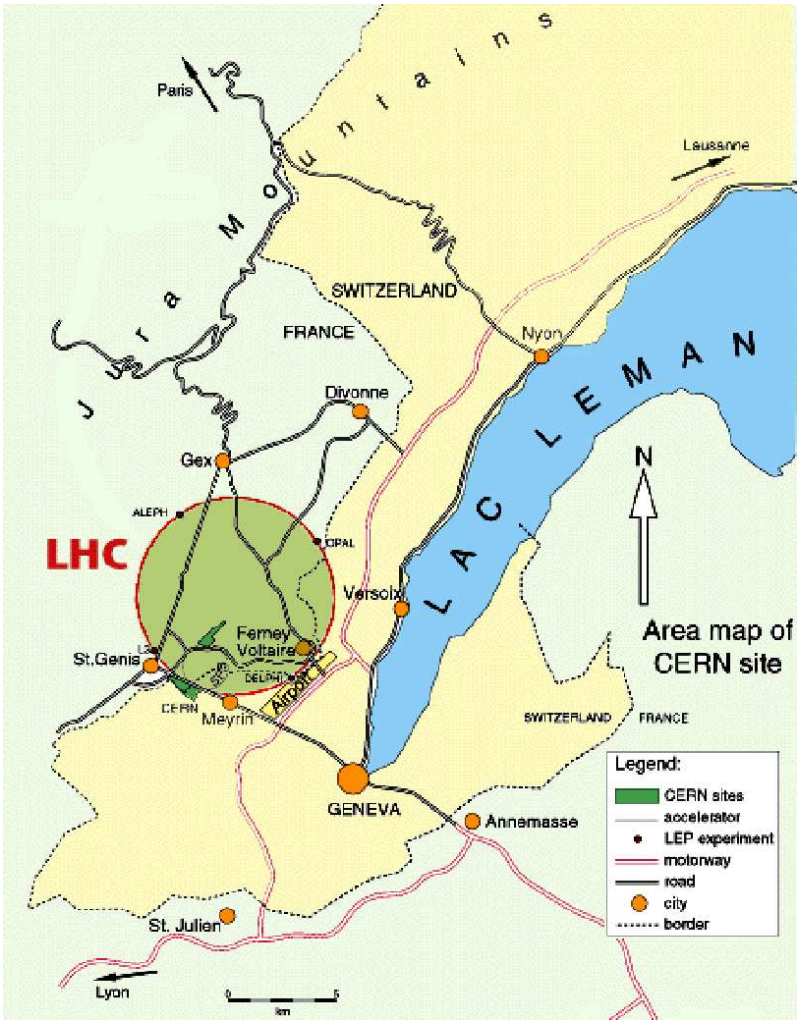
Współczesna fizyka cząstek elementarnych stawia wiele pytań dotyczących natury zjawisk zachodzący w mikroświecie, a mających decydujący wpływ na otaczającą nas rzeczywistość. *Model standardowy* - jej podstawowe obecnie narzędzie zakłada istnienie czterech oddziaływań elementarnych: elektromagnetycznych, grawitacyjnych oraz jądrowych silnych i słabych. Model ten jest dobrze przetestowaną teorią pozwalającą na wyjaśnienie licznych zjawisk, jednak nie umożliwia przeprowadzenia unifikacji oddziaływań co stanowi marzenie kolejnych pokoleń fizyków. Model standardowy nie odpowiada również na pytanie o mechanizm uzyskiwania masy przez cząstki elementarne oraz o powód ich różnorodności.

Pierwszy z w/w problemów może zostać rozwiązany przez teorie unifikacyjne: *Wielką Unifikację* oraz *Teorię Supersymetrii (SUSY)*, drugi: poprzez tzw. *hipotezę Higgsa* która zakłada istnienie skalarnego pola wypełniającego całą przestrzeń, oddziaływującego z cząstkami (poprzez nośnik tzw. *bozon Higgsa*) i nadającego im masę.

Problemem tych, jak i innych teorii współczesnej fizyki cząstek elementarnych jest ich weryfikowalność. Dziedzina ta słusznie określana jest również nazwą: „fizyka wysokich energii" - aby wniknąć wystarczająco głęboko ($10^{-19}m$) w strukturę materii potrzebna jest energia rzędu 10-15 TeV. Takowej nie posiada żaden z istniejących akceleratorów. Na

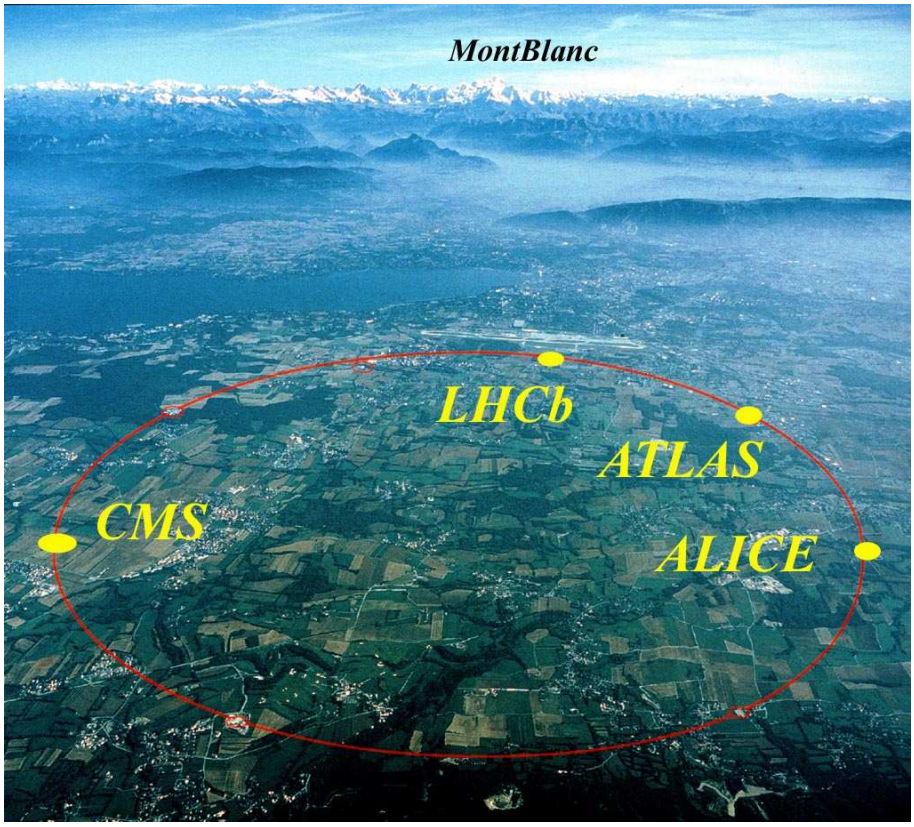
powodzenie eksperymentu ma także wpływ tzw. *świetlność* - wielkość charakteryzująca intensywność zderzających się wiązek w akceleratorze. Również odpowiedniej wartości tego parametru ($10^{34} cm^{-2} s^{-1}$) nie są w stanie zapewnić obecne narzędzia badawcze.

Dlatego też w 1994 roku Rada *CERN (fr. Conseil Européene pour la Recherche Nucléaire) - Europejskiej Organizacji Badań Jądrowych* oficjalnie zatwierdziła budowę nowego akceleratora *LHC (ang. Large Hadron Collider)* o niespotykanych dotychczas parametrach. Postanowiono w tym celu wykorzystać istniejący tunel akceleratora *LEP (ang. Large Electron-Positron Collider)* o długości 27 km znajdujący się 100 m pod powierzchnią, niedaleko Genewy (rys. 1).



Rysunek 1: Lokalizacja akceleratora LHC *(Źródło: CERN)*

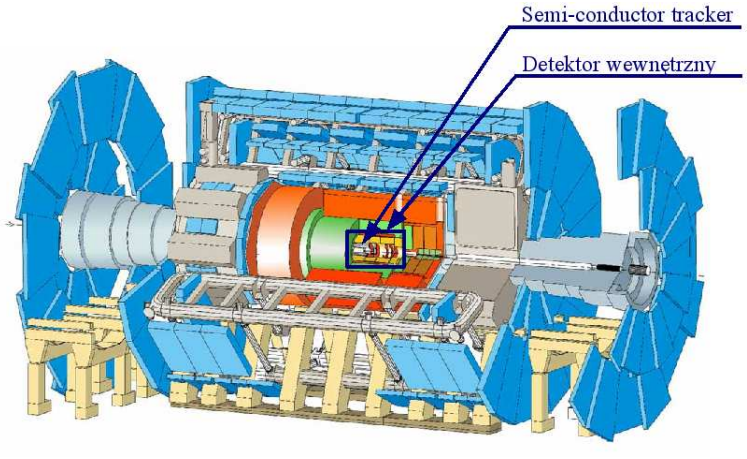
LHC, docelowo (planowane uruchomienie ma nastąpić w 2007r.) ma rozpędzać (a raczej „utrzymywać w stanie rozpędzenia") dwa „zestawy" 2838 pakietów, z których każdy zawiera 10^{11} protonów. Na pierścieniu akceleratora rozmieszczono punkty zderzeń (rys. 2) - zwane przez fizyków eksperymentami. W każdym z nich znajdzie się wyspecjalizowany system umożliwiający badanie wybranych zjawisk fizycznych związanych ze zderzeniami cząstek.



Rysunek 2: Eksperymenty akceleratora LHC i ich rozmieszczenie *(Źródło: CERN)*

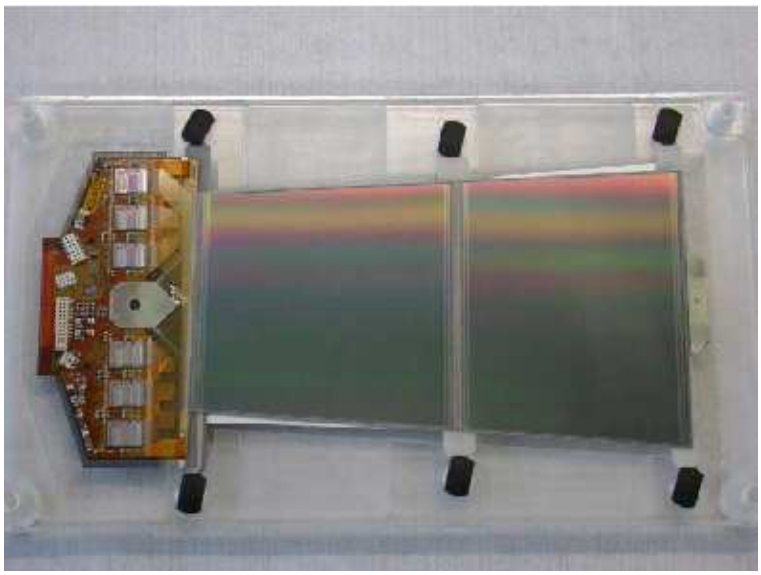
W to imponujące przedsięwzięcie techniczne ma również swój wkład grupa krakowska, skupiona wokół Instytutu Fizyki Jądrowej im. Henryka Niewodniczańskiego i Akademii Górniczo-Hutniczej. Zaangażowanie to dotyczy głównie eksperymentu ATLAS którego podstawowym zadaniem jest zweryfikowanie hipotezy Higgsa - wyselekcjonowanie bozonów Higgsa wśród powstałych produktów zderzeń. ATLAS to złożony system detektorów krzemowych, komór mionowych i kalorymetrów oraz zespół nadprzewodzących magnesów

- łącznie ok. 7000 ton różnorakiego oprzyrządowania (rys. 3)



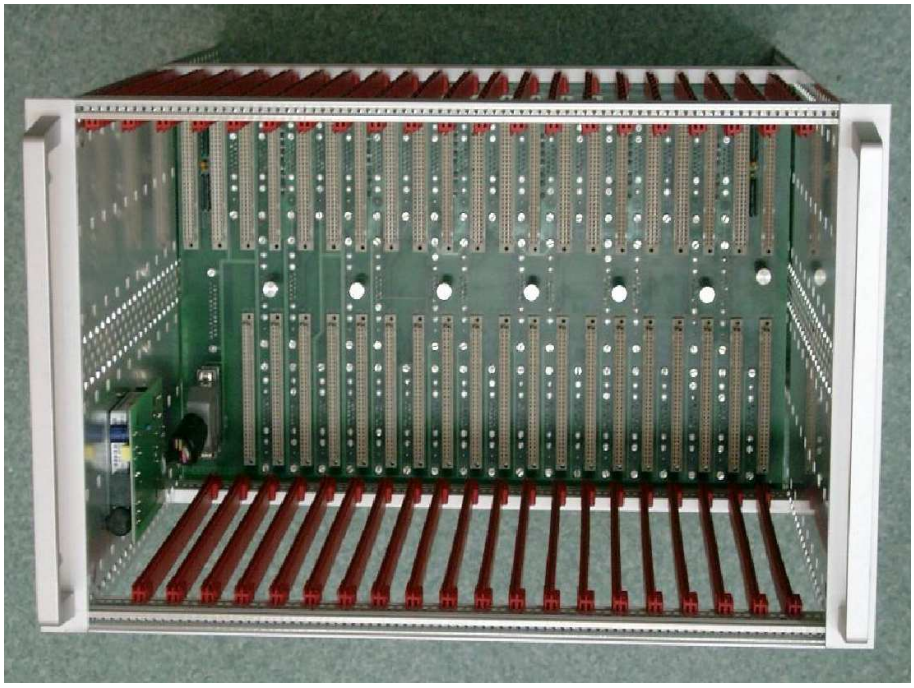
Rysunek 3: Przekrój detektora ATLAS *(Źródło: CERN)*

Krakowscy fizycy i inżynierowie zajmują się różnymi aspektami eksperymentu. Zadaniem grupy kierowanej przez prof. Piotra Maleckiego jest m.in. zrealizowanie (we współpracy z zespołem z Pragi) systemu zasilającego mikropaskowe detektory krzemowe (ang. semi-conductor tracker - rys. 4) będące częścią tzw. detektora wewnętrznego (oba zaznaczono strzałką na rys. 3).



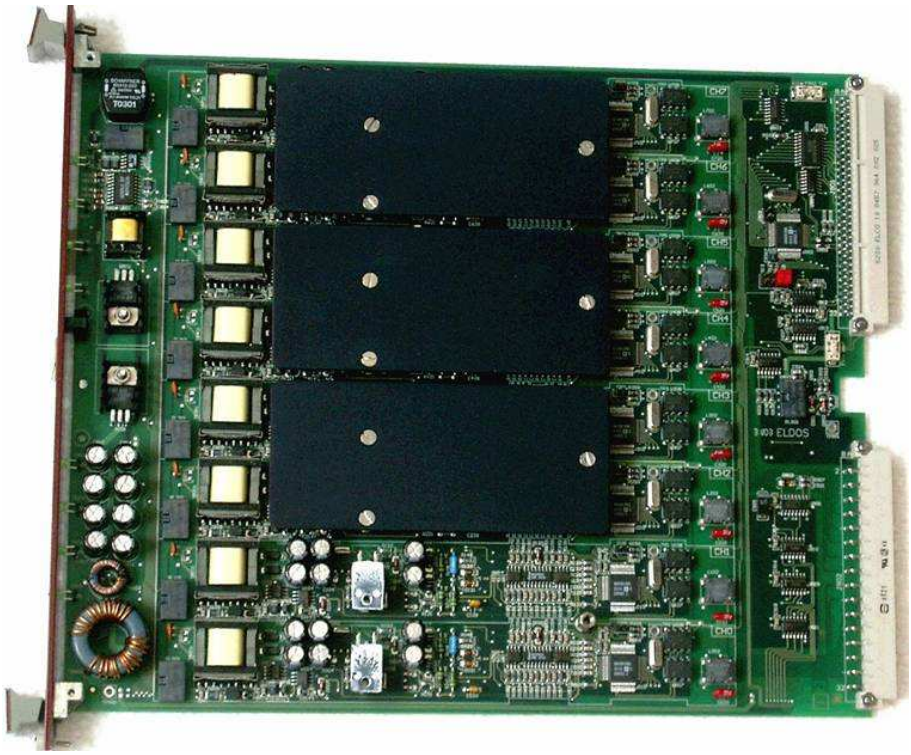
Rysunek 4: Semi-conductor tracker *(Źródło: CERN)*

Ten złożony system zasilający składa się z 511 kart wysokiego napięcia - HV (ang. High Voltage) i 1022 niskiego napięcia - LV (ang. Low Voltage) umieszczonych w kasecie (rys. 5). Pierwsze z w/w kart zaprojektowano w krakowskim IFJ, a ich produkcja odbywa się w firmie *Fideltronik Imel*, natomiast zasilacze niskiego napięcia są domeną grupy z Pragi.



Rysunek 5: Jedna z kaset systemu zasilającego SCT *(Źródło: IFJ)*

Każda karta HV (zaprezentowana na rys. 6) posiada 8 kanałów. Za sterowanie nimi odpowiadają, umieszczone na karcie, odpowiednio zaprogramowane mikrokontrolery $AD\mu C812$ firmy *Analog Devices* zwane w niniejszej pracy *kontrolerami kanałów*. Dzięki zastosowaniu dodatkowego, dziewiątego mikrokontrolera $AD\mu C812$ - tzw. *kontrolera karty* możliwa jest komunikacja z wyższą warstwą sterującą (*kontrolerem kasety*) i zadawanie różnych poziomów napięć dla każdego z wybranych kanałów.



Rysunek 6: Karta HV *(Źródło: IFJ)*

Problemami które wiążą się z zastosowaniem takiego rozwiązania są:

- rozrzut produkcyjny napięć referencyjnych użytych mikrokontrolerów - co powoduje niedokładność ustawiania napięć wyjściowych (korygowaną przez ładowanie do nich różnych wersji oprogramowania),
- 12-bitowa rozdzielczość przetwornika AC mikrokontrolera $AD\mu C812$ (ma ona również wpływ na niedokładność pracy układu),
- ekstremalne warunki w jakich pracują karty HV (mogą one powodować awarie),
- zużycie materiałów z których zbudowane są karty (mające wpływ na jakość ich funkcjonowania).

Istnieje więc potrzeba wykonania wyspecjalizowanego układu testującego opisywane urządzenia zasilające. Powinien on posiadać następujące cechy:

- możliwość wygodnego testowania prawidłowego funkcjonowania kart HV z zastosowaniem dobrze skalibrowanego, dokładniejszego przetwornika AC
- posiadanie wielu predefiniowanych trybów testowania (oraz ew. możliwość budowy własnych procedur testujących)
- możliwość ładowania oprogramowania dla mikrokontrolerów w układach zasilaczy, z automatycznym doborem wersji programu
- możliwość dokumentowania wyników procesu testowania

Układ taki (zwany również *testerem*), wraz z towarzyszącym oprogramowaniem, został zaprojektowany i wykonany w okresie: maj 2004 - maj 2005 w Instytucie Fizyki Jądrowej przez studentów Inżynierii Teleinformatycznej na Politechnice Krakowskiej: Zenona Cyganka i autora niniejszej rozprawy. Składa się on z elektronicznej płyty testującej przyłączanej do pojedynczej karty HV (w izolacji od reszty systemu zasilającego) oraz programu działającego na komputerach klasy PC, komunikującego się z płytą testera przez dwa porty szeregowe RS232C.

Praca ta, wraz z [1], stanowi kompletną dokumentację wykonanego projektu. Szczególny nacisk (zgodnie z obranym tytułem) został w niej położony na opis oprogramowania testującego (nazwanego przez autora *HV_tester*).

Rozdział 1 zawiera krótki opis kart wysokiego napięcia. Informuje o ich możliwościach i podstawach budowy. Koncentruje się głównie na charakterystyce protokołu komunikacji kontrolera kasety z kontrolerem karty, ponieważ układ testujący musi się do tego protokołu ściśle stosować pełniąc w owym „dialogu" rolę kontrolera kasety.

Rozdział 2 zawiera na wstępie krótki opis całości układu testującego. Następnie scharakteryzowano protokoły komunikacyjne przy użyciu których komputer z zainstalowanym programem *HV_tester* wymienia informację z kontrolerami karty HV oraz układem testera.

Rozdział 3 omawia podstawowe aspekty funkcjonowania programu testującego - budujące go klasy oraz ich metody i zmienne. Przedstawiony jest również interfejs użytkownika

oraz powiązania pomiędzy jego elementami a funkcjami i zmiennymi zdefiniowanymi w programie *HV_tester*.

Rozdział 4 zawiera opis implementacji testów kart wysokiego napięcia. Przedstawione są zarówno podstawowe „cegiełki" z których testy mogą być składane jak i przykłady testów już zrealizowanych. W rozdziale tym zaprezentowano również postać raportów stanowiących wynik działania procedury testującej.

Rozdział 1

Karty HV

1.1 Informacje podstawowe

Zgodnie ze specyfikacją [2] karty wysokiego napięcia użytkowane w eksperymencie ATLAS posiadają m.in. następujące cechy:

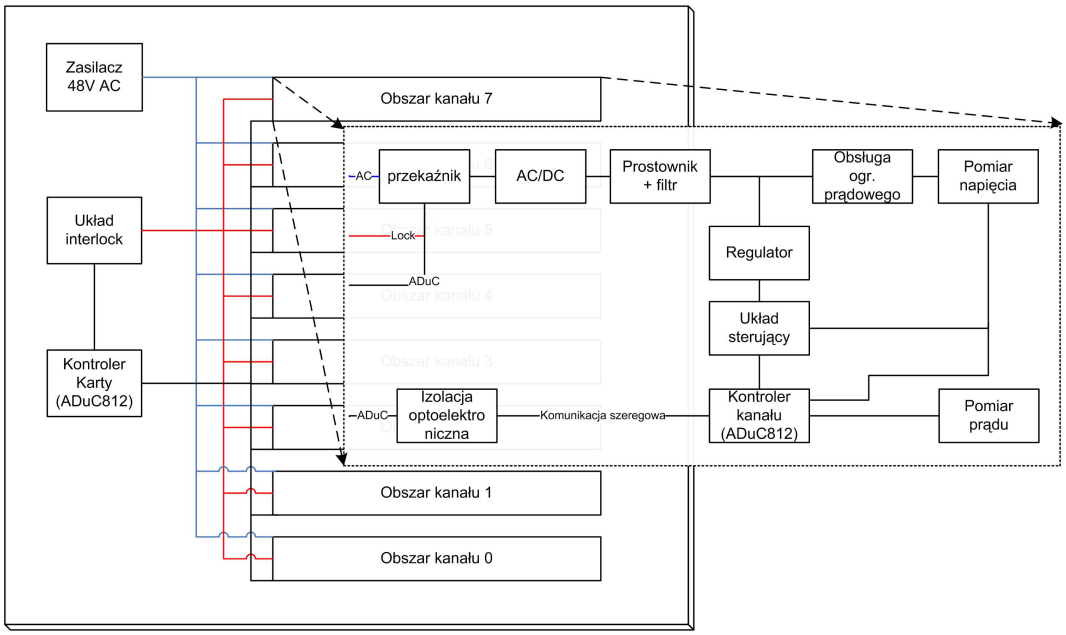
- nominalne napięcie wyjściowe w zakresie 0-500 V (górne ograniczenie stanowi zabezpieczenie napięciowe - ang. over-voltage limit) ustawiane z dokładnością większą niż 1%
- możliwość programowego ustawiania szybkości zmian napięcia (ang. ramping speed) w zakresie 5-50 V/s
- wielozakresowy (4 przełączane rezystory próbne) pomiar prądu w zakresie 10 nA - 5 mA, zabezpieczenie prądowe (ang. over-current limit) ustawiane programowo (max. 5,0 mA)
- sterowanie i odczyt na żądanie poprzez szynę CAN Bus
- możliwość sprzętowego wyłączania grup (2 razy po 4) kanałów (ang. interlock)

Karta zaprojektowana w Instytucie Fizyki Jądrowej jest płytą obwodu drukowanego zbudowanego z następujących elementów składowych:

- generator 48 V AC

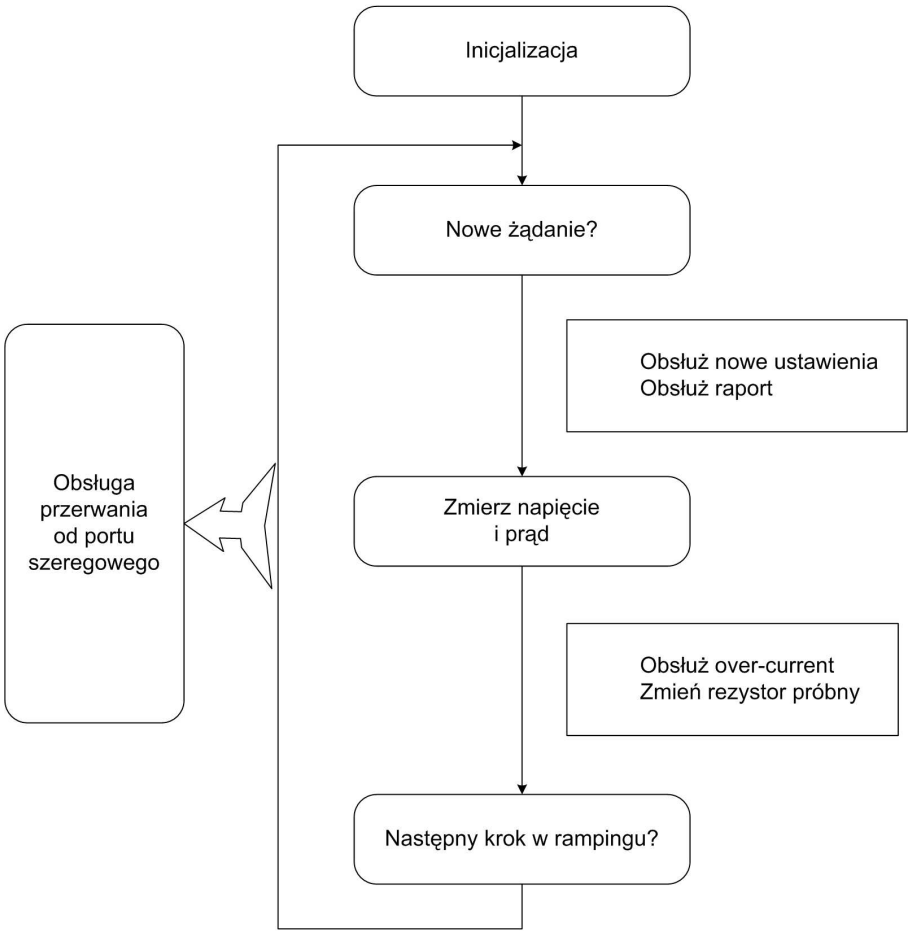
- układ interlock
- kontroler karty ($AD\mu C812$ firmy *Analog Devices*)
- 8 kanałów zasilaczy HV - każdy z nich zawiera: kontroler kanału (także w postaci mikrokontrolera $AD\mu C812$), transformator AC-DC, układ komunikacji szeregowej z kontrolerem karty (ze sprzężeniem optoelektronicznym), prostownik z filtrem, regulator napięcia, układ zapewniający zabezpieczenie prądowe oraz układy odczytu napięcia i prądu

Schemat blokowy (rys. 1.1) przedstawia w przystępny sposób powiązania pomiędzy poszczególnymi elementami karty wysokiego napięcia, wgłębiając się również w strukturę pojedynczego kanału.



Rysunek 1.1: Schemat blokowy karty HV

Podstawowym elementem każdego kanału napięciowego jest mikrokontroler sterujący, realizujący określony algorytm działania przedstawiony na rysunku 1.2.



Rysunek 1.2: Algorytm funkcjonowania kontrolera kanału

Pracuje on w pętli nieskończonej zawierającej realizację żądań, pomiar napięcia, prądu i uskutecznienie rampingu napięciowego. Pomiar prądu realizowany jest na 4 rezystorach próbnych, przełączanych programowo w zależności od zakresu w jakim pomiar ten jest dokonywany (tab. 1.1).

Pętla pracy mikrokontrolera kanału przerywana jest tylko i wyłącznie przez wywołanie obsługi przerwania.

W celu zdiagnozowania stanu każdego kanału na panelu czołowym karty umieszczono 8 diod. Stany diod wraz z ich wyjaśnieniem podano w tabeli 1.2.

Rezystor próbny	Minimalny mierzony prąd	Maksymalny mierzony prąd
$R = 62005\Omega$	$9,84nA$	$40\mu A$
$R = 12005\Omega$	$30,09\mu A$	$206\mu A$
$R = 2405\Omega$	$150\mu A$	$1031\mu A$
$R = 435\Omega$	$830,6\mu A$	$5747\mu A$

Tabela 1.1: Rezystory próbne dla pomiaru prądu kanału HV

Stan diody	Objaśnienie
wyłączona	kanał wyłączony
miga bardzo wolno	kanał włączony, nieustawiony
miga wolno	kanał włączony, napięcie 0V
miga szybko	kanał włączony, w trakcie rampingu
świeci się	kanał włączony

Tabela 1.2: Stany diod diagnostycznych

Mimo tego, że bardziej szczegółowe przedstawienie zasad działania samej karty leży poza zakresem merytorycznym niniejszej pracy należy poświęcić trochę miejsca szczególnie ważnemu z punktu widzenia syntezy układu testującego aspektowi jej funkcjonowania - komunikacji kontrolera karty z kontrolerem kasety.

1.2 Karta a kaseta

Jak wspomniano na wstępie każda z kart HV umieszczana jest w kasecie (standard 6U-280). Poprzez dwa gniazda J901 wyprowadzone są z niej kanały napięciowe (gniazdo dolne) oraz sygnały sterujące/komunikacyjne (gniazdo górne). Każda karta systemu zasilającego SCT posiada swój adres, nadawany jej po umieszczeniu w kasecie (sprzętowo, poprzez linie oznaczone BA0 - BA4) - adres ten jest wczytywany z linii przez mikrokontroler karty po jego ponownym uruchomieniu. Karty wysokiego napięcia mają przypisane miejsca (a co za tym idzie i adresy) o numerze podzielnym przez 3 z resztą 1 (tj. 1,4,7,10,...). Kontroler kasety jest zawsze umieszczany na drugiej pozycji, karta interlock - pierwszej. Pozostałe miejsca w kasecie zajmują karty niskiego napięcia (jak pokazano w tabeli 1.3).

Pozycja	$Adres_{16}$	Karta
1	0x12	Interlock
2		Kontroler kasety
3	0x00	LV
4	0x01	HV
5	0x02	LV
6	0x03	LV
7	0x04	HV
8	0x05	LV
9	0x06	LV
10	0x07	HV
11	0x08	LV
12	0x09	LV
13	0x0A	HV
14	0x0B	LV
15	0x0C	LV
16	0x0D	HV
17	0x0E	LV
18	0x0F	LV
19	0x10	HV
20	0x11	LV

Tabela 1.3: Rozmieszczenie kart w kasecie

Adresowanie kart pozwala na ich jednoznaczną identyfikację co stanowi niezbędny element skutecznej komunikacji ze światem zewnętrznym (kontrolerem kasety). Podmiotem tego „dialogu" jest kontroler karty. Jego zadaniem jest odbieranie poleceń od kontrolera kasety, sprawdzanie czy jest ich adresatem, interpretowanie i (w razie potrzeby) komunikacja (szeregowa, master-slave) z podległymi mu kontrolerami kanałów. Cały opisany powyżej proces wymiany informacji odbywa się przy użyciu linii równoległych wyprowadzonych z karty poprzez górne gniazdo J901. Ich listę wraz z funkcjami jakie pełnią przedstawiono w tabeli 1.4

„Umowa określająca format oraz znaczenie komunikatów" [3] wymienianych przez kontroler kasety i kontroler karty stanowi protokół komunikacji pomiędzy tymi urządzeniami. Do celów tworzenia oprogramowania testującego niezbędne jest zapoznanie się z nim - kluczowa jest zwłaszcza znajomość zawartości szyn danych w trakcie komunikacji równo-

Linia	Objaśnienie
D/AD 0 - D/AD7	8-bitowa szyna adresowa/danych
$\overline{DS}$	data strobe - znacznik danych (na szynie D/AD)
$\overline{AS}$	adress strobe - znacznik adresu (na szynie D/AD)
$\overline{ACK}$	acknowledgement - potwierdzenie (od kontrolera karty)
$R/\overline{W}$	read/write - znacznik odczytu/zapisu
$\overline{RESET}$	sygnał resetujący mikrokontroler karty
$\overline{BUSY}$	sygnalizacja: kontroler karty zajęty
INTERLOCK 0	sprzętowe wyłączenie kanałów 0-3
INTERLOCK 1	sprzętowe wyłączenie kanałów 4-7

Tabela 1.4: Linie komunikacji równoległej kontroler kasety-kontroler karty

ległej (formatu polecenia oraz odpowiedzi nań). Kolejne podrozdziały pracy obejmować będą omówienie „układu ramek" protokołu dla poszczególnych poleceń (zebranych w odpowiednie grupy). Pominięty zostanie (omówiony szczegółowo w [1]) aspekt komunikacji związany z odpowiednimi zmianami stanów linii sterujących ponieważ nie jest on interesujący z punktu widzenia projektu oprogramowania testera.

1.3 Karta: polecenia odczytu

Do poleceń odczytujących stan karty odbieranych i obsługiwanych przez jej kontroler należą:

- polecenie *read board (card)* (odczytanie ogólnego stanu karty),
- polecenie *read channel* (odczytanie stanu wybranego kanału znajdującego się na danej karcie)

Format danych dla obu poleceń wraz z objaśnieniem poszczególnych bajtów transmisji zaprezentowano w tabelach 1.5, 1.6.

Dane wysyłane do kontrolera karty

LP	Bajt	Wartości bitów 7-0							
1	COMMAND	0	1	X	A4	A3	A2	A1	A0
Kod polecenia(64 dziesiętnie) wraz z adresem żądanej karty									

Dane odbierane od kontrolera karty

LP	Bajt	Wartości bitów 7-0							
1	BOSTATS[0]	I47	I30	BUSY	X	X	X	X	X
Status karty: I47=1 gdy włączony interlock dla kanałów 4-7 I30=1 dla włączonego interlock 0-3 BUSY=1 gdy kontroler karty w trakcie przetwarzania									
2	BOSTATS[1]	MASK							
8 bitowa maska karty (określająca włączone kanały)									
3	BOSTATS[2]	SOFT							
Wersja oprogramowania kontrolera karty (2 półbajty w kodzie BCD)									

Tabela 1.5: Format polecenia read board

Dane wysyłane do kontrolera karty

LP	bajt	wartości bitów 7-0							
1	COMMAND	0	0	X	A4	A3	A2	A1	A0
Kod polecenia(0 dziesiętnie) wraz z adresem żądanej karty									

Dane odbierane od kontrolera karty

LP	bajt	wartości bitów 7-0							
1	BUFINHV[0]	0	0	0	0	0	1	0	1
Liczba wysyłanych przez tester bajtów (NBYTE=5_{10})									
2	BUFINHV[1]	OV	OC	PERR	UN	OFF	CNR	R1	R0
Status kanału: OV=1 gdy przekroczono zakres napięciowy (over-voltage trip) OC=1 gdy przekroczono zakres prądowy (over-current trip) PERR=1 gdy wystąpił błąd parzystości (parity-error) UN=1 gdy stan kanału niestabilny (zmiana napięcia) OFF=1 gdy kanał jest wyłączony CNR=1 gdy kanał nie odpowiada (channel not responding) R1,R0 - wybrany rezystor próbny									
3	BUFINHV[2]	VOLT_H							
4 najbardziej znaczące bity odczytanego napięcia									
4	BUFINHV[3]	VOLT_L							
8 najmniej znaczących bitów odczytanego napięcia									
5	BUFINHV[4]	CURR_H							
4 najbardziej znaczące bity odczytanego prądu									
6	BUFINHV[5]	CURR_L							
8 najmniej znaczących bitów odczytanego prądu									

Tabela 1.6: Format polecenia read channel

Uwagi:

- X - oznacza stan nieokreślony danego bitu
- Informacja na temat rezystora próbnego użytego do pomiaru prądu zawarta jest w dwóch bitach R0 i R1 obecnych w 2 bajcie odpowiedzi na *read channel* i tak dla R1,R0=00 pomiaru dokonano na rezystorze o $R = 62005\Omega$ dla bitów R1,R0=01 na rezystorze o $R = 12005\Omega$, dla bitów R1,R0=10 na rezystorze o $R = 2405\Omega$, a dla R1,R0=11 na rezystor o $R = 435\Omega$.
- 12-bitowe napięcie (V) i prąd (μA) przelicza się na wartości dziesiętne wg. następujących wzorów:

$$I = \frac{610,5 * (CURR\_H * 256 + CURR\_L)}{R} \tag{1.1}$$

$$U = \frac{VOLT\_H * 256 + VOLT\_L}{8} \tag{1.2}$$

1.4 Karta: polecenia zapisu

Do grupy poleceń które umożliwiają zmianę stanu karty zgodnie z zadanymi przez kontroler kasety parametrami należą:

- polecenie *write board (card)* (zmiana ogólnego stanu karty),
- polecenie *write channel* (ustawienie stanu wybranego kanału)

W tabelach 1.7, 1.8 zaprezentowano składnię obu poleceń, wraz z opisem bajtów wysyłanych przez kontroler kasety (w obu przypadkach kontroler karty jedynie sygnalizuje poprawność wykonania zadania zmieniając stan linii sterujących nie przesyłając żadnych informacji przez szynę danych/adresów).

Uwagi:

- Rzeczywiste napięcie U przelicza się na wartość 12-bitową wg. następujących wzorów:

$$VOLT\_H = \frac{8 * U}{256}; VOLT\_L = 8 * U\%256 \tag{1.3}$$

Dane wysyłane do kontrolera karty

LP	bajt	wartości bitów 7-0							
1	COMMAND	1	1	X	A4	A3	A2	A1	A0
Kod polecenia(192 dziesiętnie) wraz z adresem żądanej karty									
2	MASK	MASK							
Nowa, 8-bitowa maska karty (wyznacza włączone i wyłączone kanały)									

Tabela 1.7: Format polecenia write board

Dane wysyłane do kontrolera karty

LP	bajt	wartości bitów 7-0							
1	COMMAND	1	0	X	A4	A3	A2	A1	A0
Kod polecenia(128 dziesiętnie) wraz z adresem żądanej karty									
2	TOCHANN[0]	CH2	CH1	CH0	X	X	X	X	X
Numer kanału którego polecenie dotyczy (0-7)									
3	TOCHANN[1]	NBYTE							
Liczba istotnych bajtów w dalszej części transmisji (obecnie 12_{10})									
4	TOCHANN[2]	SET_COMMAND							
Kod polecenia (SET_COMMAND=32_{10})									
5	TOCHANN[3]	VOLT_H							
4 najbardziej znaczące bity ustawianego napięcia									
6	TOCHANN[4]	VOLT_L							
8 najmniej znaczących bitów ustawianego napięcia									
7,8	TOCHANN[5],[6]	TRIP_H, TRIP_L							
Ograniczenie prądowe dla rezystora $R = 62005\Omega$									
9,10	TOCHANN[7],[8]	TRIP_H, TRIP_L							
Ograniczenie prądowe dla rezystora $R = 12005\Omega$									
11,12	TOCHANN[9],[10]	TRIP_H, TRIP_L							
Ograniczenie prądowe dla rezystora $R = 2405\Omega$									
13,14	TOCHANN[11],[12]	TRIP_H, TRIP_L							
Ograniczenie prądowe dla rezystora $R = 435\Omega$									
15,16	TOCHANN[13]	RAMPING							
Ramping dla ustawianego napięcia									
17	TOCHANN[14]	MARKER							
Znacznik końca pakietu (0xEC)									

Tabela 1.8: Format polecenia write channel

- Wartości bajtów określających poziomy ograniczeń prądowych przeliczone dla poszczególnych rezystorów wyznacza się na podstawie zależności:

$$CURTRP = \frac{2 * I_{trip}}{1221} \tag{1.4}$$

jeśli $\frac{CURTRP}{256} > 15$ to:

$$TRIP\_H = FFh, TRIP\_L = FFh \tag{1.5}$$

jeśli $\frac{CURTRP}{256} < 15$ to:

$$TRIP\_H = \frac{CURTRP}{256}, TRIP\_L = CURTRP\%256 \tag{1.6}$$

(I_{trip} jest ograniczeniem prądowym w μA zadanym w postaci dziesiętnej)

- Dostępne jest 5 poziomów rampingu określonych przez bajt RAMPING polecenia *write channel*. Ma on następujące znaczenie:
  RAMPING=0 oznacza brak rampingu
  RAMPING=1 oznacza ramping na poziomie 50 V/s
  RAMPING=2 oznacza ramping na poziomie 20 V/s
  RAMPING=3 oznacza ramping na poziomie 10 V/s
  RAMPING=4 oznacza ramping na poziomie 5 V/s

1.5 Karta: polecenia inne

Ostatnią grupą poleceń o której należy wspomnieć są polecenia „bezramkowe" tj. realizowane tylko i wyłącznie poprzez zmianę stanu odpowiedniej linii sterującej. Do grupy tej należą rozkazy:

- *set interlock*
- *reset*

Pierwsze z wyżej wymienionych poleceń polega na wymuszeniu grupowego wyłączenia kanałów 0-3 i/lub 4-7 poprzez zmianę stanu linii *INTERLOCK 0* i/lub *INTERLOCK 1* na stan wysoki. Ponowne uruchomienie kanału jest możliwe dopiero po przywróceniu stanu niskiego na odpowiedniej linii *INTERLOCK* oraz zadanie maski (polecenie *write board*) w której bit odpowiadający numerowi kanału posiada ustawioną wartość 1.

Polecenie *reset* wymusza ponowne uruchomienie kontrolera karty. Jego wykonanie ogranicza się jedynie do zmiany stanu linii $\overline{RESET}$ dochodzącej do danej karty. Obecność polecenia *reset* jest szczególnie warta nadmienienia, ze względu na fakt, że po procesie ładowania oprogramowania do kontrolera karty oraz po zmianie adresu samej karty wykonanie operacji *reset* jest niezbędne do zadziałania wprowadzonych zmian. Będzie więc ono intensywnie wykorzystywane w zaprojektowanym układzie testującym. Kolejny rozdział pracy pokrótce przedstawi ideę jego funkcjonowania, skupiając się przede wszystkim na programie testującym *HV_tester*.

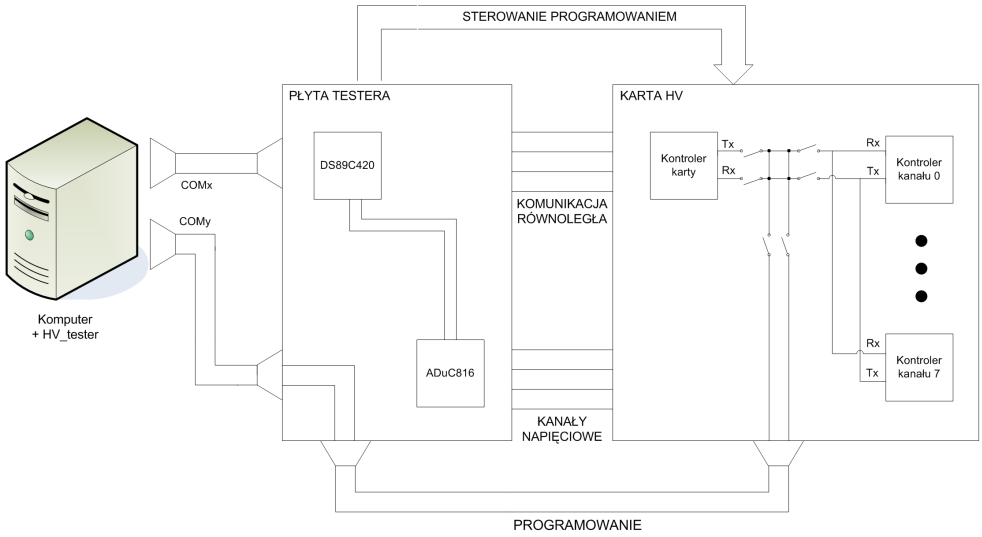
Rozdział 2

Idea funkcjonowania oprogramowania testującego

2.1 Informacje podstawowe - współpraca z układem testującym

Jak wspomniano na wstępie zaprojektowany układ testujący składa się z płyty obwodu drukowanego (szczegółowo omówionej w [1]) oraz programu testującego pracującego na komputerze klasy PC. Schemat 2.1 pokazuje poglądowo te elementy, ilustrując również sposób ich współfunkcjonowania.

Płyta testera przyłączana jest do karty HV poprzez gniazda J901 nań umieszczone. Komunikacja z kartą odbywa się przez linie równoległe przy zastosowaniu się do zasad protokołu kontroler kasety-kontroler karty omówionego szczegółowo w poprzednim rozdziale pracy (oraz w [1]). Na płycie umieszczone są dwa mikrokontrolery: DS89C420 firmy Dallas-Maxim oraz $AD\mu C816$ produkcji Analog Devices. Pierwszy (tzw. mikrokontroler główny) zawiaduje komunikacją PC-tester-karta HV oraz umożliwia przełączenie kontrolerów kasety/karty w tryb programowania. Drugi (tzw. mikrokontroler pomiarowy lub pomocniczy), zaopatrzony jest w precyzyjny, skalibrowany przetwornik AC który pozwala na dokładny pomiar napięcia i prądu każdego z kanałów.



Rysunek 2.1: Schemat ideowy układu testującego

Program testujący *HV_tester* współpracuje z testerem korzystając z dwóch portów szeregowych RS232C (opisy charakteryzujące szczegółowo ten bardzo popularny standard transmisji danych można znaleźć w [4] oraz [5]). Pierwszy z nich pozwala na komunikację (wg. uzgodnionego protokołu) z mikrokontrolerem DS89C420 (wydawanie mu poleceń i odbieranie danych). Drugi umożliwia przeprowadzenie transmisji programującej wybrany mikrokontroler karty HV (uprzednio przełączony w tryb programowania przez mikroprocesor DS89C420). Naturalną koleją rzeczy następne części pracy omawiać będą wymienione aspekty interakcji komputera z zainstalowanym programem *HV_tester* z resztą układu testującego oraz kartą wysokiego napięcia.

2.2 Protokół komunikacji PC - układ testujący

Wymiana informacji pomiędzy płytą testera a komputerem osobistym odbywa się przy prędkości 9600 bitów na sekundę, 8 bitach danych, z jednym bitem stopu i bez kontroli parzystości oraz przepływu. Transmisja realizowana jest w trybie master-slave - „mistrzem",sterującym jej przebiegiem, jest komputer.

W pojedynczej operacji wymiany danych pomiędzy podmiotami transmisji można wy-

różnić następujące fazy:

1. Przesłanie polecenia wraz z jego argumentami (komputer do testera) - ogólny format pakietu danych polecenia przedstawiono w tabeli 2.1. Zawiera on kod polecenia (patrz: Dodatek A), dane oraz sumę kontrolną liczoną jako 8-bitową sumę bez przeniesienia wszystkich bajtów transmisji.

LP	Bajt
1	CODE
Kod polecenia dla układu testującego	
2...n	DATA
Bajty danych	
(n+1)	CHECKSUM
Suma kontrolna	

Tabela 2.1: Format pakietu polecenia

2. Przesłanie potwierdzenia otrzymania/realizacji polecenia (tester do komputera). Faza ta następuje po wykonaniu żądania zawartego w poleceniu (jeśli oczywiście polecenie to jest prawidłowe i możliwe jest jego wykonanie). Potwierdzeniem jest jeden bajt ACK (ang. acknowledge) o wartości zależnej od statusu wykonanej operacji lub przyczyny dla której zaniechano jej realizacji (tab. 2.2).

Oznaczenie	Bajt_{10}	Opis
ACK_OK	170	Polecenie otrzymane/zrealizowane
ACK_CRC_ERROR	204	Błąd sumy kontrolnej
ACK_BUSY_ERROR	240	Karta HV zajęta (sygnał BUSY)
ACK_CARD_DEAD	15	Karta HV nie odpowiada
ACK_ADUC_DEAD	85	Mikrokontroler pomiarowy testera nie odpowiada

Tabela 2.2: Kody potwierdzeń i ich znaczenie

3. Wysłanie pakietu danych będących wynikiem funkcjonowania polecenia (tester do komputera) - jeśli realizacja danego polecenia owe dane generuje. Ogólny format pakietu w tej fazie zaprezentowano w tabeli 2.3.

LP	Bajt
1...m	DATA
Bajty danych odpowiedzi	
(m+1)	CHECKSUM
Suma kontrolna	

Tabela 2.3: Format pakietu odpowiedzi

Wśród poleceń można wyróżnić dwie grupy:

1. Polecenia których ostatecznym adresatem jest tester.
2. Polecenia których ostatecznym adresatem jest kontroler karty/kanału.

Do pierwszej należą polecenia:

- *reset* - polecenie to powoduje wymuszenie stanu niskiego na linii $\overline{RESET}$ (tak jak opisano to w podrozdziale 1.5) co wywołuje ponowne uruchomienie kontrolera karty. Zlecenie wykonania polecenia *reset* polega na wysłaniu do układu testującego tylko dwóch bajtów: kodu polecenia i sumy kontrolnej (tab. 2.4).

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	1	1	1	0	0	0	1
Kod polecenia *reset*									
2	CHECKSUM	0	1	1	1	0	0	0	1
Suma kontrolna									

Tabela 2.4: Format polecenia *reset*

Odpowiedź na polecenie reset składa się tylko i wyłącznie z bajtu potwierdzenia ACK.

- *set board address* - umożliwia ustawienie adresu testowanej karty (tester wymusza odpowiedni stan linii *BA0-BA4*). Operacja *SBA* wykonywana jest na początku procesu testowania, ponieważ adres ten jest niezbędnym argumentem poleceń których adresatem jest kontroler karty. Ramkę komendy *set board address* przedstawia tab. 2.5.

LP	bajt	wartości bitów 7-0							
1	CODE	0	1	1	0	0	1	1	1
Kod polecenia sba									
2	ADDRESS	X	X	X	A4	A3	A2	A1	A0
Nadawany adres karty (5 najmłodszych bitów)									
3	CHECKSUM								
Suma kontrolna									

Tabela 2.5: Format polecenia *set board address*

Po otrzymaniu potwierdzenia dotyczącego pozytywnej realizacji *set board address* program testujący uruchamia ponownie kontroler karty (zlecając *reset*) aby odczytał swój nowy adres z linii *BA0-BA4*.

- *read tester status* - polecenie (tab. 2.6) umożliwiające odczyt stanu testera (wprowadzone głównie w celu sprawdzenia czy tester jest w ogóle przyłączony do wybranego portu szeregowego komputera).

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	1	1	0	0	0	0	1
Kod polecenia rts									
2	CHECKSUM	0	1	1	0	0	0	0	1
Suma kontrolna									

Tabela 2.6: Format polecenia *read tester status*

Po otrzymaniu polecenia mikrokontroler główny testera sprawdza czy obecny jest mikrokontroler pomocniczy i odsyła, w zależności od wyniku tego sprawdzenia, ACK jeśli $AD\mu C816$ funkcjonuje poprawnie i ACK_ADUC_DEAD w przeciwnym przypadku. Rozmiar odsyłanej (po ACK) paczki danych (m=16 lub 8 bajtów) również zależy od tego, czy nawiązana została komunikacja z mikrokontrolerem pomocniczym - odpowiedź układu testującego (tab. 2.7) zawiera ciąg „ADuC816+DS89C420" lub „DS89C420".

LP	Bajt
1...m	DATA
Dane dot. statusu testera	
(m+1)	CHECKSUM
Suma kontrolna	

Tabela 2.7: Format odpowiedzi na *read tester status*

- *chip programming* - polecenie (tab. 2.8) umożliwiające przestawienie kontrolerów karty/kanału w tryb programowania (opisane w szczególe w kolejnym podrozdziale i w pracy [1]) oraz wyjście z tegoż trybu po załadowaniu programu. Argument OP_TYPE określa rodzaj operacji i tak:
  OP_TYPE=0 gdy programowany mikrokontroler ma powrócić do normalnej pracy. Wymaga to wykonania (po otrzymaniu potwierdzenia ACK dla *chip programming*)

operacji zresetowania kontrolera karty (jeśli programowany był mikroprocesor karty) lub ponownego uruchomienia kontrolera kanału (poprzez zadanie odpowiedniej maski poleceniem *write board*) gdy ładowano nowy program do mikroprocesora sterującego kanałem HV.

OP_TYPE=2 gdy programowany ma być mikrokontroler karty,

OP_TYPE=1 gdy nowy program ma zostać załadowany do kontrolera kanału - program testujący po otrzymaniu potwierdzenia musi dokonać wyboru programowanego kanału poprzez operację wyłączenia pozostałych mikrokontrolerów zasilaczy - polecenie *write board* (w tryb programowania przestawiane są wszystkie układy sterujące zasilaczami ale program ładowany jest tylko do jednego z nich).

LP	bajt	wartości bitów 7-0							
1	CODE	0	1	1	1	0	0	0	0
Kod polecenia chp									
2	OP_TYPE	0	0	0	0	0	0	T1	T0
Rodzaj operacji programującej									
3	CHECKSUM								
Suma kontrolna									

Tabela 2.8: Format polecenia *chip programming*

- *set interlock* - polecenie (tab. 2.9) wymuszające ustawienie (przez układ testera) linii *INTERLOCK 0* i/lub *INTERLOCK 1* dochodzącej do karty HV w żądany stan, co powoduje albo wyłączenie (stan wysoki linii) albo włączenie (stan niski) kanałów 0-3 i/lub 4-7. Odpowiedzią układu testującego na polecenie *set interlock* jest tylko i wyłącznie bajt ACK informujący o pozytywnym jego zrealizowaniu lub ewentualnym błędzie.

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	1	1	0	1	0	0	1
Kod polecenia si									
2	LOCK	I47	I30	0	0	0	0	0	0
Żądany stan interlocków									
3	CHECKSUM								
Suma kontrolna									

Tabela 2.9: Format polecenia *set interlock*

- *prepare external measurement* - polecenie przygotowujące ustrój pomiarowy umieszczony na płycie testera do wykonania pomiaru prądu i/lub napięcia wybranego kanału. Polecenie to ze względów bezpieczeństwa powinno być wykonywane przy zerowym napięciu kanału na którym ma zostać dokonany pomiar (w programie testującym *HV_tester* kanał ten jest po prostu wyłączany). Odpowiedzią testera na polecenie *prepare external measurement* jest wyłącznie bajt ACK.

LP	bajt	wartości bitów 7-0							
1	CODE	0	0	1	1	1	0	1	0
Kod polecenia pem									
2	MASK	M7	M6	M5	M4	M3	M2	M1	M0
Kanał (tylko jeden!) x wybrany do pomiaru $\Leftrightarrow$ Mx=1									
3	CHECKSUM								
Suma kontrolna									

Tabela 2.10: Format polecenia *prepare external measurement*

- *external read voltage* - polecenie umożliwiające odczyt napięcia kanału (uprzednio „przygotowanego” poprzez wykonanie *prepare external measurement*). Pomiar dokonywany jest przy użyciu mikrokontrolera pomocniczego - w związku z tym posiada

on zwiększoną (do 16 bitów) rozdzielczość. Format polecenia oraz otrzymanej (po potwierdzeniu) odpowiedzi zawierającej napięcie badanego kanału przedstawiono w tab.2.11

Uwaga: napięcie w postaci dziesiętnej uzyskuje się na podstawie wzoru:

$$U = \frac{VHIGH * 256 + VLOW}{128} \tag{2.1}$$

Dane wysyłane do testera

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	1	0	0	0	0	1	0
Kod polecenia erv									
2	CHECKSUM	0	1	0	0	0	0	1	0
Suma kontrolna									

Format odpowiedzi

LP	Bajt	Wartości bitów 7-0
1	VHIGH	
8 bardziej znaczących bitów napięcia		
2	VLOW	
8 mniej znaczących bitów napięcia		
3	CHECKSUM	
Suma kontrolna		

Tabela 2.11: Format polecenia *external read voltage* i odpowiedzi nań

- *external read current* - polecenie pozwalające na zmierzenie prądu „przygotowanego" kanału (również poprzez *prepare external measurement*) przy zadanym obciążeniu. Format ramki *external read current* oraz odpowiedzi którą w wyniku tego polecenia odsyła układ testujący przedstawiono w tabeli 2.12.

Dane wysyłane do testera

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	1	0	0	0	0	1	0
Kod polecenia erc									
1	LOAD	RL7	RL6	RL5	RL4	RL3	RL2	RL1	0
Obciążenie przy którym ma zostać wykonany pomiar									
3	CHECKSUM	0	1	0	0	0	0	1	0
Suma kontrolna									

Format odpowiedzi

LP	Bajt	Wartości bitów 7-0							
1	CURRH								
8 bardziej znaczących bitów prądu									
2	CURRL								
8 mniej znaczących bitów prądu									
3	MESTATUS	0	0	0	0	0	0	R1	R0
Użyty rezystor próbny R: $R1 = 0, R0 = 0 \Leftrightarrow R = 74883\Omega$ $R1 = 0, R0 = 1 \Leftrightarrow R = 14924\Omega$ $R1 = 1, R0 = 0 \Leftrightarrow R = 2981,1\Omega$ $R1 = 1, R0 = 1 \Leftrightarrow R = 563,94\Omega$									
4	CHECKSUM								
Suma kontrolna									

Tabela 2.12: Format polecenia *external read current* i odpowiedzi nań

Uwagi:

- Wynik pomiaru można przeliczyć na wartość dziesiętną zgodnie ze wzorem 2.2.

$$I = \frac{62,5 * (CURRH * 256 + CURRL)}{R} - offset(R) \tag{2.2}$$

offset(R) przyjmuje wartości:
offset(74883)=0,66
offset(14924)=0,41
offset(2981,1)=0,27
offset(563,94)=-2,4
i reprezentuje przesunięcie dobranej doświadczalnie charakterystyki przetwarzania AC dla każdego z rezystorów próbnych (patrz: [1]).

- Rezystory obciążające odpowiadające poszczególnym bitom maski LOAD mają wartości:
RL7=1 $G\Omega$
RL6=100 $M\Omega$
RL5=20 $M\Omega$
RL4=8,6 $M\Omega$
RL3=940 $k\Omega$
RL2=100 $k\Omega$
RL1=10 $k\Omega$
Wartość 1 bitu RLx maski LOAD oznacza że rezystor RLx jest włączony, a 0 - wyłączony.
- Łączna rezystancja obciążenia jest sumą wszystkich rezystancji obciążających (połączenie szeregowe).
- Przełączanie rezystora próbnego odbywa się automatycznie w zależności od zakresu w jakim pomiar jest dokonywany (patrz: [1])

- *set 5W resistors* - polecenie to włącza/wyłącza rezystory obciążające 100 kΩ dla kanałów zadanych 8 bitową maską stanowiącą drugi bajt wysyłanej ramki (tab. 2.13).

Obecnie w programie testującym rezystory te nie są wykorzystywane. Odpowiedzią na omawiane polecenie jest jedynie bajt potwierdzenia ACK.

LP	Bajt	Wartości bitów 7-0							
1	CODE	0	0	1	0	0	1	0	1
Kod polecenia s5Wr									
2	MASK5W	M7	M6	M5	M4	M3	M2	M1	M0
Maska określająca kanały dla których rezystor 5W ma zostać włączony (stan 1)									
3	CHECKSUM								
Suma kontrolna									

Tabela 2.13: Format polecenia *set 5W resistors*

Drugą grupę poleceń wysyłanych przez program *HV_tester* do układu testującego formują opisane już w poprzednim rozdziale komendy protokołu kaseta-karta:

- *read board*
- *read channel*
- *write board*
- *write channel*

Aby ułatwić mikrokontrolerowi testera zadanie przekazania ramek w/w poleceń do ich ostatecznych adresatów zastosowano kapsułkowanie (ang.encapsulation) [3] tj. są one „obudowane" jedynie nagłówkiem i sumą kontrolną (jak pokazano w tabeli 2.14), a tester nie modyfikuje ich zawartości. Nagłówek stanowi wyłącznie kod polecenia.

LP	Bajt
1	CODE
Kod polecenia	
2...k	CARRIED_BYTES
Przenoszone bajty protokołu kaseta-karta	
(k+1)	CHECKSUM
Suma kontrolna	

Tabela 2.14: Kapsułkowanie protokołu kaseta-karta

Warto w tym miejscu nadmienić, że poszczególne polecenia otrzymały różne kody, ze względu na odmienne akcje jakie tester musi podjąć po ich przekazaniu do kontrolera karty (np. *read channel* wymaga przygotowania się na odbiór 6 bajtów danych, *write channel* - jedynie na potwierdzenie wykonania zadania).

Oczywiście kapsułkowanie dotyczy również odpowiedzi na komendę - w sytuacji gdy takowa występuje i zawiera istotne dane (*read channel*, *read board*). W tym przypadku jednak do oryginalnej ramki dodawana jest, zgodnie z ogólnymi wytycznymi protokołu podanymi na początku niniejszego rozdziału, jedynie suma kontrolna.

2.3 Protokół komunikacji programującej

Programowanie mikrokontrolerów ADuC812 znajdujących się na płytach zasilaczy jest jedną z podstawowych funkcji projektowanego testera. Programowanie to odbywa się przez port szeregowy, wg. ustalonego schematu postępowania - opisanego szczegółowo w nocie technicznej [6].

Pierwszą czynnością którą należy wykonać jest przestawienie mikrokontrolera w stan programowania - poprzez wymuszenie stanu niskiego na nóżce $\overline{PSEN}$ i przez czas co najmniej 10ms [11] nóżce $\overline{RESET}$ procesora (czyli jego zresetowanie). Operacje te, zarówno w przypadku kontrolera kanału jak i kontrolerów karty, są wykonywane przez układ testera (po otrzymaniu przezeń polecenia *chip programming* zmieniany jest stan nóżki $\overline{PSEN}$,

potem program testujący nakazuje zresetowanie mikrokontrolera karty (*reset*) lub włączenie i wyłączenie kanału (*write board*).Od tego momentu mikrokontroler pracuje w trybie ładowania programu (ang. loader mode).

W trybie tym port szeregowy mikrokontrolera o danej częstotliwości zegara działa z prędkością określoną wzorem (2.3) z 8 bitami danych i bez kontroli parzystości.

$$BAUDRATE = \frac{9600 * czestotliwość\_zegara}{11,0592Mhz} \tag{2.3}$$

W związku z tym dla kontrolera karty którego częstotliwość zegara wynosi 16Mhz szybkość transmisji programującej wynosi 13889 bitów/s , a dla mikrokontrolera zawiadującego kanałem (zegar 8Mhz) ma ona wartość 6944 bitów/s.

Na wstępie procesu programowania loader „przedstawia się" wysyłając 25 bajtowy pakiet identyfikacyjny zawierający identyfikator mikrokontrolera, wersję loadera i konfigurację. Zabezpieczony jest on 1-bajtową sumą kontrolną.

Samo programowanie odbywa się poprzez transmisję pakietów przedstawionych w tabeli 2.15. Pierwsze dwa bajty pakietu stanowi identyfikator (PACKET ID = 0x07 0x0E),

LP	Bajt
1,2	PACKET ID
2 bajty identyfikacyjne	
3	BYTES_NO
Ilość bajtów danych (włączając polecenie)	
4	COMMAND
Kod polecenia dla loadera	
5...m	DATA
0-25 bajtów danych	
(m+1)	CHECKSUM
Suma kontrolna	

Tabela 2.15: Pakiet protokołu programowania mikrokontrolerów firmy Analog Devices

następny bajt definiuje ilość następujących bajtów ramki (1-25), kolejny (COMMAND) - rodzaj polecenia:

COMMAND = 'C' - wykasowanie pamięci programu

COMMAND = 'A' - wykasowanie pamięci programu i danych

COMMAND = 'W' - zaprogramuj blok pamięci programu

COMMAND = 'E' - zaprogramuj blok pamięci danych

COMMAND = 'U' - wykonaj skok do danego adresu w pamięci i wykonaj znajdujący się tam program

po nim następują dane stanowiące zwykle część ładowanego programu (jeśli dane polecenie ich wymaga ; jako źródło danych zawierających program stosowany jest standardowy plik Intel Hex), a na koniec wysyłana jest suma kontrolna zabezpieczająca integralność transmisji [6]. Dodatkowo, dla zwiększenia bezpieczeństwa komunikacji, każde pozytywnie zrealizowane polecenie jest potwierdzane przez loader poprzez przesłanie bajtu ACK (06h), a w przypadku jakichkolwiek błędów wysyłany jest Negative-ACK (07h).

Wyjście z trybu programowania odbywa się poprzez przywrócenie pierwotnego stanu nóżki $\overline{PSEN}$ i zresetowanie mikrokontrolera.

Kolejny rozdział pracy poświęcony będzie wykonanej implementacji programu testującego *HV_tester* - zostanie w nim również przedstawiony sposób realizacji zarówno komunikacji z układem testującym jak i transmisji programującej.

Rozdział 3

Realizacja programu testującego

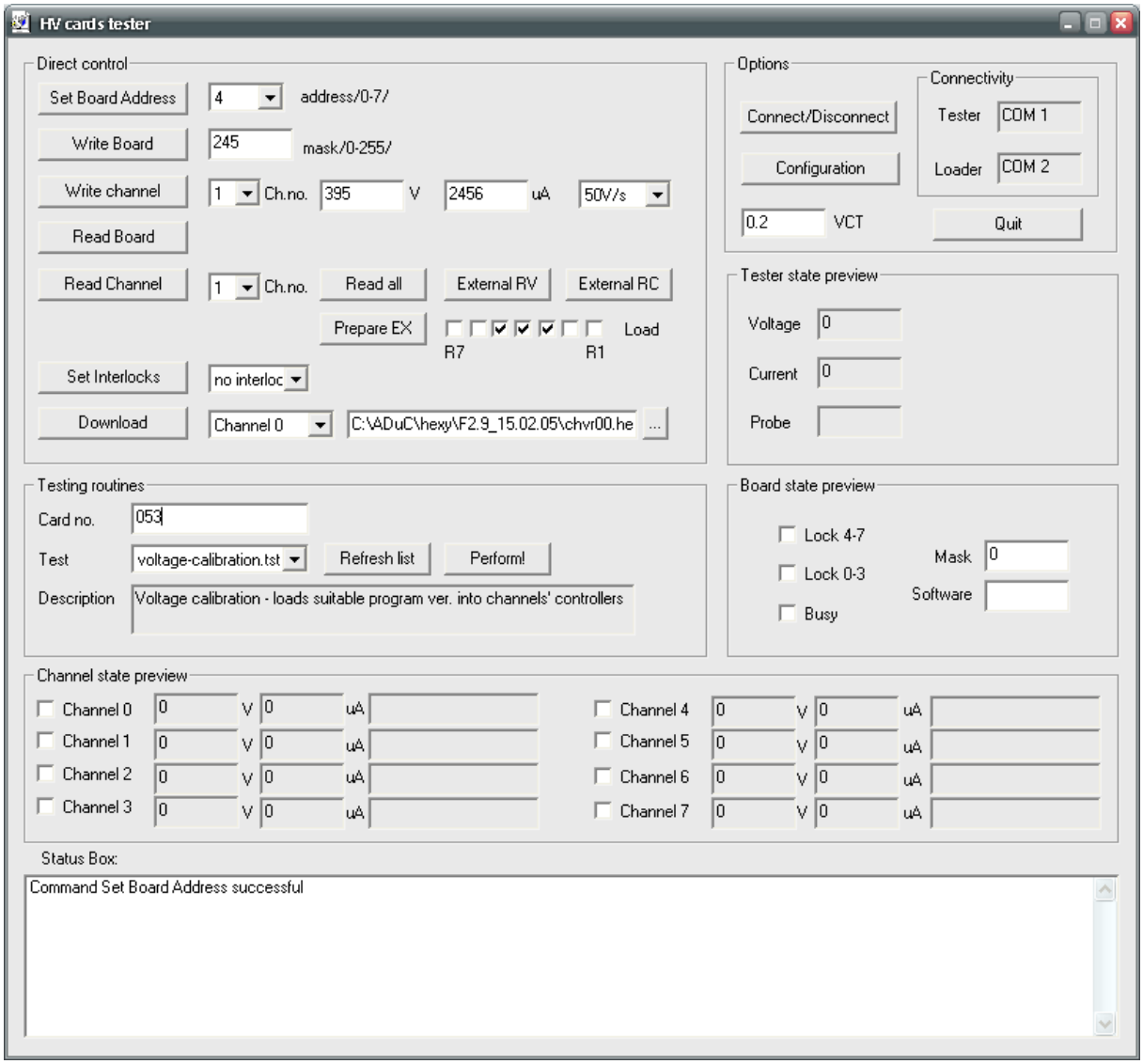
Program testujący *HV_tester* został wykonany w języku C++, w oparciu o środowisko Microsoft Visual Studio 6.0 pracujące pod kontrolą systemu operacyjnego Microsoft Windows. Wybór platformy został częściowo podyktowany ogólną dostępnością rozwiązań firmy z Redmond. Kluczowym argumentem za „implementacją okienkową" była jednak możliwość ustawiania niestandardowych prędkości pracy portu szeregowego w systemie Windows, nawet przy zastosowaniu konwerterów USB-RS232C (funkcji tej nie obsługują jeszcze niektóre sterowniki rozważanej alternatywy - Linuksa). Częstotliwości taktowania mikrokontrolerów karty i kanału wymuszają użycie owych nietypowych prędkości (jak opisano szczegółowo w podrozdziale 2.3) - ograniczając tym samym możliwość dowolnego wyboru systemu operacyjnego pod kontrolą którego program testujący miał funkcjonować.

Środowisko Visual C++ wyróżnia się łatwością programowania interfejsu przy użyciu biblioteki MFC (Microsoft Foundation Classes), bardzo dobra przenośność skompilowanego kodu, a i obsługa portu szeregowego nie nastręcza żadnych większych trudności - w związku z tym zostało ono wybrane jako narzędzie implementacyjne (przy wyborze brano również pod uwagę alternatywę w postaci środowiska C++ Builder firmy Borland).

Program pozwala zarówno na zadawanie pojedynczych komend testujących kartę HV za pomocą przyjaznego interfejsu użytkownika jak i odczyt i realizację plików testów (domyślne rozszerzenie *tst*, domyślny katalog *tests*) opisanych szczegółowo w Rozdziale 4. Podwaliny interfejsu stworzone zostały przy użyciu kreatora aplikacji (ang. App Wizard)

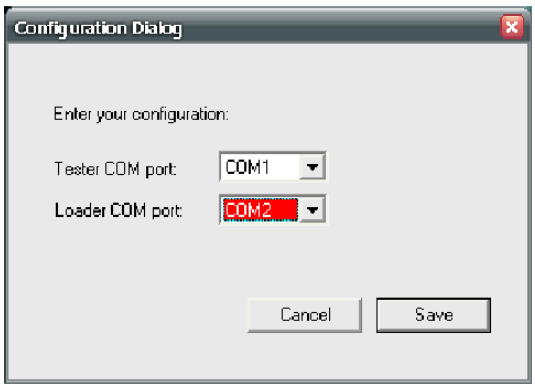
zgodnie z instrukcjami zawartymi w [7]. Dalsze operacje wykonywane były przez „ręczną modyfikację" kodu źródłowego programu. Interfejs użytkownika aplikacji składa się z 4 okien dialogowych:

- okno główne /zasób IDD_HV_TESTER_DIALOG/ - przedstawione na rys. 3.1, zapewnia użytkownikowi podstawową interakcję z układem testującym, realizację prostych poleceń ale i również wybór oraz wykonanie (w oknie testów) złożonego zestawu komend zapisanych w pliku testu



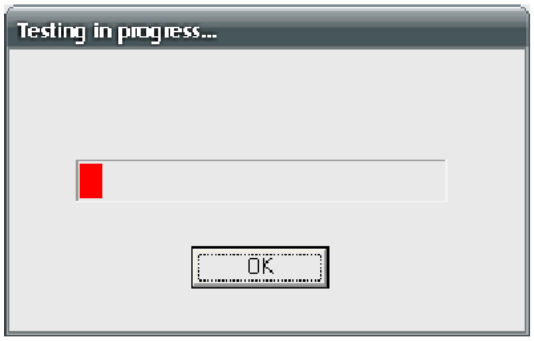
Rysunek 3.1: Okno główne programu HV_tester

- okno konfiguracyjne /zasób IDD_CONFDIALOG/ - zilustrowane na rys. 3.2, wywoływane na początku działania programu jeśli nie znaleziono pliku konfiguracyjnego lub na żądanie użytkownika (z głównego okna dialogowego), pozwala na wybór portów do jakich przyłączony jest układ testujący



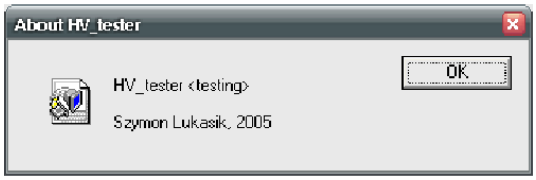
Rysunek 3.2: Okno konfiguracyjne programu HV_tester

- okna testów /zasób IDD_TEST_DIALOG/ - pokazane na rys.3.3, okno to informuje użytkownika o postępie testu zadanego z pliku o rozszerzeniu *tst*



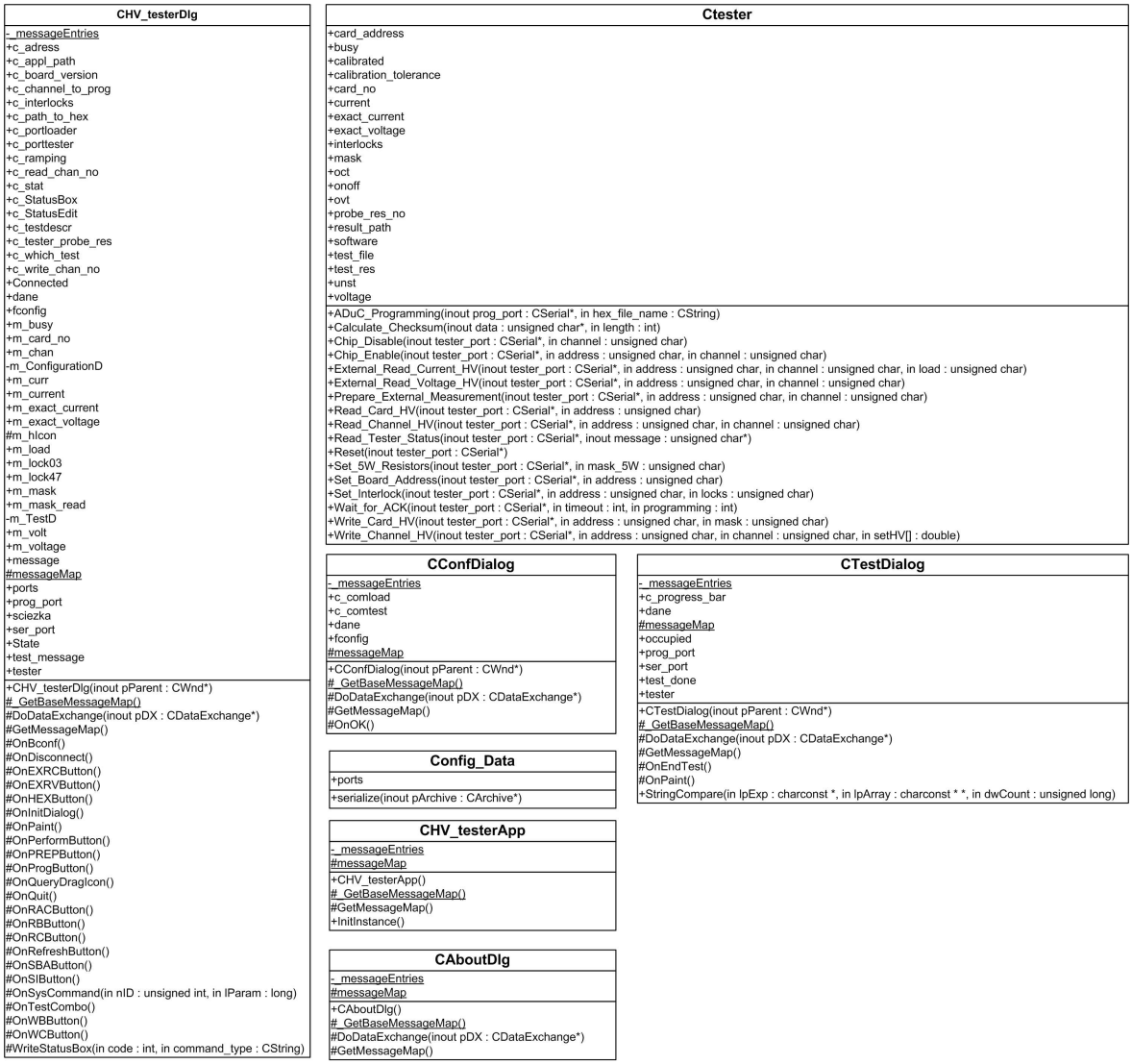
Rysunek 3.3: Okno testów programu HV_tester

- okna informacyjnego /zasób IDD_ABOUTBOX/ - okno to (rys.3.4) zawiera informacje na temat wersji programu oraz jego autora



Rysunek 3.4: Okno informacyjne programu HV_tester

HV_tester jest aplikacją obiektową - specyfikację klas aplikacji przedstawiono na rys. 3.5.



Rysunek 3.5: Specyfikacja klas programu HV_tester

W kolejnych podrozdziałach zostaną pokrótce omówione funkcje jakie poszczególne klasy pełnią w programie testującym, ich metody i zmienne składowe oraz sposób interakcji z zasobami okien dialogowych.

3.1 Klasa *CSerial*

Klasa *CSerial* (źródło: [8]) obsługuje komunikację niskiego poziomu - przesyłanie i odbieranie bajtów przez port szeregowy. Zapewnia ona następujące metody:

- `BOOL Open( int nPort, int nBaud )` - funkcja otwierająca port szeregowy `nPort` (numeracja zgodna z oznaczeniami portów COM w systemie Microsoft Windows) z prędkością `nBaud`. Zwraca `TRUE` jeśli operacja zakończyła się sukcesem, `FALSE` w przeciwnym przypadku. Wszystkie kolejne metody wymagają, by port skojarzony z *CSerial* (poprzez wykonanie `Open`) był otwarty.

- `int SendData( unsigned char *buffer, int size )` - funkcja wysyłająca `size` kolejnych bajtów danych, wskazanych przez `buffer`. Wynikiem zwracanym przez tą metodę jest liczba pomyślnie przesłanych bajtów.

- `int ReadDataWaiting( void )` - funkcja sprawdzająca zawartość bufora odbioru portu szeregowego i zwracająca liczbę bajtów oczekujących w nim na odczytanie.

- `int ReadData( void *buffer, int limit )` - metoda ta odczytuje maksymalnie `limit` bajtów z bufora odczytu portu szeregowego. Dane zapisywane są do tablicy wskazanej przez `buffer`. Funkcja zwraca rzeczywistą ilość odczytanych bajtów.

- `BOOL Close( void )` - metoda umożliwiająca zamknięcie portu szeregowego dotąd skojarzonego z klasą *CSerial*. Zwraca `TRUE` jeśli operacja ta udała się, a `FALSE` w przypadku wystąpienia problemów (np. gdy dany port nie został wcześniej otwarty).

3.2 Klasa *Ctester*

Klasa ta jest główną klasą programu. Jej listing zamieszczono w Dodatku B. Metody *Ctester* pozwalają na realizację wszystkich zadań komunikacyjnych *HV_testera*, a zmienne określają stan karty HV i układu testera w danej chwili czasu (są one uaktualniane tylko w przypadku udanej operacji odczytu).

Zmienne składowe klasy *Ctester* to:

- `unsigned char card_address` - adres karty nadany jej w wyniku działania *set board address*
- `unsigned char mask,software,busy` - aktualna maska karty HV, wersja oprogramowania kontrolera karty i stan linii busy (odczytane przez *read board*)
- `BOOL interlocks[2]` - stan linii interlocków dla testowanej karty (uzyskiwany również w wyniku działania polecenia *read board*)
- `double voltage[8], double exact_voltage[8]` - wartości napięć poszczególnych kanałów, zmierzone odpowiednio: przez kontrolery kanałów (*read channel*) i zewnętrzny przetwornik pomiarowy (*external read voltage*).
- `double current[8], double exact_current[8]` - prądy wszystkich kanałów zmierzone w wyniku działania poleceń *read channel* i *external read current*
- `BOOL onoff[8], BOOL ovt[8], BOOL oct[8], BOOL unst[8]` - stany bitów OFF, OVT, OCT, UNST odczytane poleceniem *read channel* dla każdego z kanałów
- `unsigned char probe_res_no[8]` - numer rezystora próbnego na którym dokonano ostatniego pomiaru prądu każdego z kanałów (*read channel*)
- `unsigned char test_res` - rezystor próbny użyty do ostatniego pomiaru prądu przy wykorzystaniu polecenia *external read current*
- `CString test_file, CString card_no, CString result_path` - zmienne dotyczące testów (opisanych szczegółowo w kolejnym rozdziale pracy): pierwsza informuje o nazwie pliku z którego został wczytany test, druga - o numerze testowanej

karty, a trzecia - o ścieżce w której ma zostać zapisany raport podsumowujący wynik testu

- `BOOL calibrated[8], double calibration_tolerance` - specyfikują parametry kalibracji napięciowej kanałów (patrz: Rozdział 4), pierwsza zmienna informuje o fakcie, że dany kanał jest już wykalibrowany, a druga - o tolerancji z jaką kalibracja ta ma być wykonywana (współczynnik VCT - patrz: Podrozdział 4.1).

Implementację funkcji komunikacyjnych programu *HV_tester* stanowią następujące metody klasy *Ctester*:

- `unsigned char Calculate_Checksum(unsigned char *data,int length)` - funkcja wywoływana przez inne metody w celu wyznaczenia sumy kontrolnej danych przeznaczonych do wysłania układowi testującemu lub danych właśnie odeń odebranych. Jako parametry wywołania `Calculate_Checksum` podawane są: wskaźnik do tablicy znakowej zawierającej dane do zsumowania oraz ilość bajtów dla których suma ma zostać policzona. Wyjściem z funkcji jest bajt obliczonej sumy kontrolnej.

- `int Wait_for_ACK(CSerial *tester_port,int timeout,BOOL programming= FALSE)` - funkcja ta weryfikuje czy otrzymano potwierdzenie od układu testującego podłączonego do komputera przez port stowarzyszony z klasą `*tester_port`. Jeśli przez czas `timeout` (podawany w milisekundach) potwierdzenie to nie nadejdzie funkcja zwraca kod błędu 4. Inne kody zwracane przez polecenia komunikacyjne zebrano w tab. 3.1. Polecenia *Wait_for_ACK* dotyczą kody 0,1,2,3,4,13 w niej zaprezentowane. Zmienna `programming` informuje czy oczekiwanie ma dotyczyć transmisji programującej. Jeśli tak, to funkcja powinna zwracać pozytywną odpowiedź - 0 także dla bajtu potwierdzenia 06h używanego w protokole transmisji programującej mikrokontrolery firmy Analog Devices (jak to opisano w Rozdziale 2 niniejszej pracy).

  Funkcja `Wait_for_ACK` jest wykorzystywana przez wszystkie inne metody komunikacyjne klasy *Ctester* (zaprezentowane poniżej) w celu weryfikacji, czy układ testujący poprawnie zinterpretował i wykonał dane polecenie. W przypadku gdy zwraca ona

Kod zwracany$_{10}$	Przyczyna
0	Brak błędów
1	Otrzymano ACK_CRC_ERROR
2	Otrzymano ACK_BUSY_ERROR
3	Otrzymano ACK_CARD_DEAD
4	Nie otrzymano ACK
5	Błąd sumy kontrolnej (po stronie komputera)
6	Timeout (upłynął czas na odpowiedź)
7	Problem z dostępem do portu szeregowego
8	Otrzymano ACK_ADuC_DEAD
11	Problem z otwarciem pliku Intel HEX (programowanie)
12	Zła struktura pliku Intel HEX (programowanie)
13	Otrzymano NACK (programowanie)

Tabela 3.1: Kody zwracane przez funkcje komunikacyjne

kod błędu różny od 4 transmisja jest ponawiana (limit powtórzeń definiuje stała RETRIES). Jeśli żaden z bajtów ACK do programu testującego nie dotarł (co odpowiada kodowi 4) samo oczekiwanie na potwierdzenie jest ponawiane (maksymalnie RETRIES razy).

- `int Read_Tester_Status(CSerial *tester_port, unsigned char *message)` - metoda zapewniająca pełną implementację polecenia *read tester status*, wraz z weryfikacją otrzymania ACK (poprzez wywołanie `Wait_for_ACK`) oraz odebraniem odpowiedzi układu testującego i zapisaniem jej do tablicy znakowej wskazanej przez `message`. Funkcja zwraca kody 0,1,2,3,4,5,6,7.

- `int Reset(CSerial *tester_port)` - funkcja wykonująca reset kontrolera karty (poprzez wprowadzenie linii RESET w stan niski na czas co najmniej 10ms). Zwraca kody: 0,1,4,7.

- `int Set_Board_Address(CSerial *tester_port, unsigned char address)` - metoda realizująca operację *set board address*. Nadawany karcie adres wskazywany jest przez zmienną `address`. Po otrzymaniu od układu testera potwierdzenia ustawienia linii adresu w żądany stan program testujący wykonuje funkcję `Reset`. Polecenie

`Set_Board_Address` zwraca kody: 0,1,4,7, a w przypadku pozytywnego wykonania uaktualnia zmienną `card_address` (przypisując jej wartość `address`).

- `int Set_Interlock(CSerial *tester_port, unsigned char address, unsigned char locks)` - funkcja wykonująca ustawienie zadanego interlock'u (zmienna `locks` tożsama z bajtem LOCK ramki *set interlock* przesuniętym w prawo o 7 bitów) dla karty o adresie `address`). Wartości zwracane przez tą metodę to: 0,1,4,7.

- `int Write_Card_HV (CSerial *tester_port, unsigned char address,unsigned char mask)` - metoda implementująca operację *write board*. Zadaną maskę podaje się jako argument `mask`. Funkcja ta zwraca wartości: 0,1,2,3,4,7.

- `int Read_Card_HV (CSerial *tester_port, unsigned char address)` - metoda ta realizuje odczyt stanu karty. Jej parametry zapisywane są do odpowiednich zmiennych klasy *Ctester*. Kody zwracane przez `Read_Card_HV` to: 0,1,2,3,4,5,7, a uaktualniane przezeń zmienne to: `mask, software, busy, interlock[]`

- `int Read_Channel_HV(CSerial *tester_port, unsigned char address, unsigned char channel)` - funkcja wykonująca operację odczytu stanu kanału *read channel* o numerze `channel`. Parametry kanału zapisywane są do odpowiednich zmiennych klasy *Ctester* omówionych powyżej (`voltage[], current[], onoff[], ovt[], oct[], unst[] i probe_res_no[]`). `Read_Channel_HV` zwraca kody: 0,1,2,3,4,5,7.

- `int Write_Channel_HV(CSerial *tester_port, unsigned char address, unsigned char channel, double setHV[3])` - metoda zapewniająca obsłużenie zapisu nowych ustawień kanału `channel` zawartych w buforze `setHV`. Jego pierwszy element to żądane napięcie kanału w V, drugi - ograniczenie prądowe w μA, a trzeci - kod rampingu o wartościach identycznych jak przedstawione w Rozdziale 1.4 pracy. Kody zwracane przez `Write_Channel_HV` to: 0,1,2,3,4,7.

- `int Chip_Enable(CSerial *tester_port,unsigned char address, unsigned char channel)` - funkcja zapewniająca realizację polecenia *chip pro-*

gramming z kodem typu (OP_TYPE) równym 1 lub 2 (rozpoczęcie programowania kontrolera kanału/karty). Jeśli parametr `channel` ma wartość 8 to wykonywana jest procedura przygotowania do programowania mikrokontrolera karty obejmująca przesłanie ramki *chip programming* (OP_TYPE=2, z oczekiwaniem na ACK) oraz wywołanie funkcji `Reset`. Jeśli jako argument `channel` podano liczbę z zakresu 0-7 to wysyłana jest ramka *chip programming* z kodem OP_TYPE równym 2, a po otrzymaniu potwierdzenia wybrany kanał jest ponownie uruchamiany (poprzez wykonanie funkcji `Write_Channel_HV` z maską zawierającą 1 tylko na pozycji odpowiadającej programowanemu kanałowi). Metoda `Chip_Enable` zwraca kody: 0,1,2,3,4,7.

- `int ADuC_Programming(CSerial *prog_port,CString file)` - funkcja realizująca programowanie mikrokontrolerów ADuC8xx zgodnie z wytycznymi podanymi w Podrozdziale 2.3. Parametr `file` wskazuje nazwę pliku Intel HEX użytego jako źródło ładowanego programu, a `prog_port` - port przez który ma zostać wykonana operacja programowania. Funkcja zwraca kody 0,4,5,7,11,12,13:

- `int Chip_Disable(CSerial *tester_port,unsigned char channel)` - metoda pozwalająca na zakończenie procesu programowania (OP_TYPE=0) kanału/karty (rozróżnienie podobnie jak w funkcji `Chip_Enable` wykonywane jest na podstawie zmiennej `channel`). W przypadku gdy funkcja ta dotyczy kontrolera karty po przesłaniu ramki *chip programming* i otrzymaniu potwierdzenia wykonywane jest reset mikrokontrolera (metoda `Reset`). Dla ADμC812 sterującego kanałem również należy, po otrzymaniu potwierdzenia, uruchomić ponownie mikrokontroler - odbywa się to poprzez dwukrotne zadanie nowej maski karty (`Write_Card_HV`): za pierwszym razem jest ona zerowa, a za drugim zawiera 1 na pozycji odpowiadającej numerowi programowanego mikrokontrolera. Metoda `Chip_Disable` zwraca kody 0,1,2,3,4,7.

- `int Prepare_External_Measurement(CSerial *tester_port, unsigned char address, unsigned char channel)` - funkcja przygotowująca ustrój pomiarowy do przeprowadzenia pomiaru napięcia/prądu kanału `channel` - implementująca ope-

rację *prepare external measurement*. Zwraca kody: 0,1,2,3,4,7.

- `int External_Read_Voltage_HV(CSerial *tester_port, unsigned char address, unsigned char channel)` - metoda realizująca *external read voltage* - odczyt napięcia kanału `channel` przy użyciu 16-bitowego przetwornika AC wbudowanego w mikrokontroler pomocniczy testera. Napięcie to zapisywane jest w zmiennej `exact_voltage[]` należącej do klasy *Ctester*. Funkcja `External_Read_Voltage` zwraca kody: 0,4,5,6,7,8.

- `int External_Read_Current_HV(CSerial *tester_port, unsigned char address, unsigned char channel,unsigned char load)` - metoda realizująca 16-bitowy pomiar prądu kanału `channel` przy obciążeniu `load` (patrz: tab.2.12). Dane o prądach zmierzonych tą metodą przechowywane są w tablicy `exact_current[]` klasy *Ctester*. Funkcja zwraca kody: 0,4,5,6,7,8.

- `int Set_5W_Resistors(CSerial *tester_port,unsigned char mask_5W)` - funkcja ta realizuje operację *set 5W resistors* - ustawienie 5W rezystorów obciążających dla kanałów określonych przez `mask_5W` (patrz: tabela 2.13).
  Metoda `Set_5W_Resistors` zwraca kody: 0,1,4,7.

3.3 Klasy *CHV_testerApp* i *CAboutDlg*

Klasa *CHV_testerApp* jest klasą wyprowadzoną z CWinApp i stanowi klasę aplikacji programu testującego. W przesłoniętej funkcji składowej CWinApp *InitInstance()* tworzony jest nowy obiekt klasy CHV_testerDlg.

Klasa *CAboutDlg* jest klasą wyprowadzoną z CDialog, powiązaną z zasobem IDD_ABOUTBOX poprzez mechanizm DDX (ang. Dialog Data Exchange) opisany szczegółowo w [9]. Zapewnia wyświetlenie okna dialogowego oraz możliwość jego zamknięcia (metoda CDialog::OnOk()).

3.4 Klasa *CHV_testerDlg*

Klasa *CHV_testerDlg* jest klasą wyprowadzoną z CDialog. Poprzez mechanizm DDX zmienne i metody tej klasy są powiązane z elementami zasobu głównego okna programu *HV_tester* (rys. 3.1). W tabeli 3.2 przedstawiono metody przypisane poszczególnym przyciskom zasobu IDD_HV_TESTER_DIALOG.

LP	Zasób	Metoda	Akcja metody
1	IDC_SBABUTTON	`OnSBAButton()`	`Ctester.Set_Board_Address()`
2	IDC_WBBUTTON	`OnWBButton()`	`Ctester.Write_Card_HV()`
3	IDC_WCBUTTON	`OnWCButton()`	`Ctester.Write_Channel_HV()`
4	IDC_RBBUTTON	`OnRBButton()`	`Ctester.Read_Card_HV()`
5	IDC_RCBUTTON	`OnRCButton()`	`Ctester.Read_Channel_HV()`
6	IDC_RACBUTTON	`OnRACButton()`	`Ctester.Read_Channel_HV()` (iteracyjnie)
7	IDC_PREPBUTTON	`OnPREPButton`	`Ctester.Prepare_External_M...()`
8	IDC_EXRCBUTTON	`OnEXRCButton`	`Ctester.External_Read_Current()`
9	IDC_EXRVBUTTON	`OnEXRVButton()`	`Ctester.External_Read_Voltage()`
10	IDC_SIBUTTON	`OnSIButton()`	`Ctester.Set_Interlock()`
11	IDC_HEXBUTTON	`OnHEXButton()`	wybierz plik HEX
12	IDC_PROGBUTTON	`OnProgButton()`	`Ctester.Chip_Enable()` `Ctester.ADuC_Programming()` `Ctester.Chip_Disable()`
13	IDC_REFBUTTON	`OnRefreshButton()`	odśwież listę testów
14	IDC_TESTBUTTON	`OnPerformButton()`	wykonaj wybrany test (`CTestDialog.DoModal()`)
15	IDOK	`OnOk()`	wyjdź
16	ID_DISCONNECT	`OnDisconnect()`	otwarcie/zamknięcie portu `tester.Read_Tester_Status()`
17	IDC_BCONF	`OnBconf()`	odczyt konfiguracji (`ConfigData.Serialize()`) ew. `CConfDialog.DoModal()`

Tabela 3.2: Powiązania przycisków IDD_HV_TESTER_DIALOG z metodami klasy *CHV_testerDlg*

Wybrane metody *OnButtonXXX* odwołują się do odpowiednich funkcji klasy *Ctester* uaktualniając w wyniku swego działania zarówno kontrolki okna dialogowego jak i zmienne tej klasy.

Klasa *CHV_testerDlg* jest również silnie powiązana z pozostałymi klasami programu. W jej metodzie *OnInitDialog()* do menu okna dialogowego dodawany jest element „About" umożliwiający dostęp do okna informacyjnego programu. Sprawdzana jest również obecność (i odczytywana zawartość - patrz 3.5) pliku konfiguracyjnego *HV_Config.dat* który powinien znajdować się w katalogu roboczym programu testującego. W przypadku jakichkolwiek problemów z zastosowaniem zawartej w nim konfiguracji wywoływana jest metoda *DoModal()* omówionej poniżej klasy *CConfDialog* (DoModal() dla okna dialogowego powoduje jego wyświetlenie). Metoda ta jest również wywoływana po naciśnięciu na zasób IDC_BCONF (przycisk „Configuration"). Po zdefiniowaniu nowych parametrów pracy konfiguracja jest odczytywana ponownie. Także dla klasy *CTestDialog* omówionej w Podrozdziale 3.6 metoda *DoModal()* jest wywoływana bezpośrednio z *CHV_testerDlg* (konkretnie z *CHV_testerDlg.OnPerformButton()*) co pozwala na przeprowadzenie testu z wybranego w oknie głównym pliku testującego.

3.5 Klasy *ConfigData* i *CCConfDialog*

Klasa *ConfigData* zawiera zmienne konfiguracyjne programu testującego (porty: testera i programujący). Są one zapisywane i odczytywane z pliku *HV_Config.dat* przy pomocy metody *Serialize()* tej klasy. Wykorzystany tu został mechanizm serializacji MFC omówiony szczegółowo w [8] i w [10].

Klasa *CConfDialog* jest klasą wyprowadzoną z CDialog i powiązaną poprzez mechanizm DDX z oknem konfiguracyjnym IDD_CONFDIALOG. Zmienne tej klasy reprezentują wartości kontrolek ustawiających porty szeregowe użytkowane przez program testujący. Ich stan jest kopiowany do elementów klasy *ConfigData* po wykonaniu metody *OnOk()* - kliknięcie na przycisk „Save" (w przypadku naciśnięcia przycisku „Cancel" okno zamykane jest standardową metodą klasy CDialog - OnCancel(), bez zapisu stanu konfiguracji).

3.6 Klasa *CTestDialog*

CTestDialog jest również pochodną klasy okna dialogowego (CDialog). W jej metodzie *OnPaint()* realizowany jest odczyt kolejnych linii pliku tekstowego zawierającego test (wybrany w oknie głównym programu). Wykonanie owego testu następuje linia po linii, zaś wizualizację tego procesu stanowi pasek postępu. Po zakończeniu tej operacji uruchamiany jest program *Notatnik* (ang. Notepad), będący standardowym edytorem tekstowym MS Windows, z otwartym plikiem raportu podsumowującego wykonany test. Okno testów można zamknąć klikając na przycisk „Ok" wywołując tym samym metodę OnOk().

Kolejny rozdział pracy przedstawi w szczególe sposób realizacji testów w programie *HV_tester*, objaśni strukturę pliku testu oraz zaprezentuje raport z jego wykonania.

Rozdział 4

Testy kart HV

Jak wspomniano w poprzednim rozdziale program testujący umożliwia realizację testów odczytywanych z plików testów o rozszerzeniu *tst*. Nie istnieje przy tym ograniczenie wyboru ze ściśle określonej, niemożliwej do rozszerzenia listy testów - pliki te są plikami tekstowymi mającymi charakter „skryptowy", umieszczanymi w podkatalogu *tests* programu *HV_tester*. Pozwala to na ich łatwą modyfikację oraz tworzenie własnych testów w ramach predefiniowanych poleceń.

Plik *tst* programu *HV_tester* posiada z góry zdefiniowaną strukturę. Pierwsza jego linia powinna zawierać identyfikacyjny ciąg znaków `HV_TESTER_TEST_FILE`. Druga stanowi opis testu (wyświetlany w oknie głównym programu). Kolejne stanowią ciąg poleceń testujących - każde z nich w pojedynczej linii. Wyjątkiem jest przypadek, gdy linia zaczyna się od znaku `#` - jest ona wtedy traktowana jako komentarz i zawarty w jej kolejnych kolumnach tekst nie jest interpretowany jako polecenie testujące.
Ogólna postać każdej linii poleceń przedstawia się następująco:

```
COMMAND PARAMETERS DESIRED_CODE OPTIONS
```

Pole `COMMAND` zawiera ciąg znaków odpowiadający jednemu z kilkunastu predefiniowanych poleceń, `PARAMETERS` to zestaw argumentów wymaganych przez wybrane polecenie, `DESIRED_CODE` definiuje kod zwracany przez polecenie który jest uznawany przez użytkow-

nika za poprawny (kody te mają identyczne znaczenia jak te podane w tab. 3.1). Ostatnie z pól - `OPTIONS` określa dodatkowe parametry pracy i może przyjmować wartości :
`BREAK` - przerywa procedurę testującą jeśli kod powrotu żądany przez użytkownika jest różny od zwracanego przez dane polecenie
`CALIBRATE` - dotyczy tylko polecenia EXREADV i informuje program testujący, że jest ono użytkowane w trybie kalibracji napięciowej (patrz 4.1).

Kolejny podrozdział pracy zawiera omówienie dostępnego zestawu poleceń - „bloków funkcjonalnych" z których można składać złożone procedury testujące.

4.1 Polecenia testujące

Program *HV_tester* umożliwia użycie w plikach *tst* następujących poleceń testujących:

- `SBA`

  Składnia: `SBA address desired_code option`

  Realizuje operację *set board address*, nadając adres `address` testowanej karcie. Wywoływana jest w tym celu metoda `Set_Board_Address()` klasy *Ctester*. Argument `address` jest wykorzystywany w kolejnych krokach procedury testującej (występuje w niemal wszystkich z poniżej zaprezentowanych poleceń).

- `WBOARD`

  Składnia: `WBOARD address mask desired_code option`

  Wykonuje ustawienie nowej maski `mask` (postać dziesiętna) karty o adresie `address`. Operacja ta odbywa się przez wywołanie funkcji `Ctester.Write_Card_HV()`.

- `RBOARD`

  Składnia: `RBOARD address proper_result desired_code option`

  Przeprowadza odczyt stanu maski (*read board*) poprzez wywołanie metody `Read_Card_HV()` klasy *Ctester*. Właściwy wynik definiuje w postaci dziesiętnej argument `proper_result`.

- `WINTERLOCK`

  Składnia: `WINTERLOCK address lock_no desired_code option`

  Wykonuje ustawienie interlocków, wywołując w tym celu funkcję `Set_Interlock()` klasy *Ctester*. Argument `lock_no` przyjmuje wartości 0,1,2,3 odpowiednio dla żądania: brak interlocków, interlock dla kanałów 0-3, interlock dla kanałów 4-7 i interlock dla wszystkich kanałów

- `RINTERLOCK`

  Składnia: `RINTERLOCK address proper_result desired_code option`

  Przeprowadza odczyt stanu interlocków (użyta w tym przypadku funkcja to `Ctester.Read_Card_HV()`), prawidłowy wynik operacji określa argument `proper_result` (o identycznym znaczeniu jak `lock_no` w poleceniu `WINTERLOCK`).

- `WCHANNEL`

  Składnia: `WCHANNEL address channel voltage current ramping desired_code option`

  Wykonuje ustawienie napięcia `voltage` kanału `channel` (przy użyciu odpowiedniej metody klasy *Ctester*), z żądanym ograniczeniem prądowym `current` i rampingiem `ramping`.

- `ROVT`

  Składnia: `ROVT address channel proper_result desired_code option`

  Sprawdza czy kanał `channel` został wyłączony w wyniku działania zabezpieczenia napięciowego (gdy `proper_result`=1) lub czy kanał ten pracuje normalnie (gdy `proper_result`=0). Weryfikacja stanu kanału odbywa się przy użyciu metody `Read_Channel_HV()` klasy *Ctester*.

- `ROCT`

  Składnia: `ROCT address channel proper_result desired_code option`

  Sprawdza czy kanał `channel` został wyłączony w wyniku działania zabezpieczenia prądowego (gdy `proper_result`=1) lub czy kanał ten pracuje normalnie (jeśli `proper_result`=0). Stan kanału jest odczytywany przy użyciu metody

`Read_Channel_HV()` klasy *Ctester*.

- `RCURVOLT`

  Składnia: `RCURVOLT address channel desired_code option`

  Wykonuje pomiar napięcia i prądu kanału `channel` przy zastosowaniu metody `Read_Channel_HV()` (napięcie i prąd odczytywane przez kartę).

- `RPROBE`

  Składnia: `RPROBE address channel proper_result desired_code option`

  Wykonuje operację odczytu stanu kanału `channel` (metoda `Read_Channel_HV()`) i sprawdza czy dziesiętna postać 2 bitów oznaczenia rezystora użytego próbnego do pomiaru prądu jest równa `proper_result` (tj. czy pomiar zrealizowano na prawidłowym rezystorze)

- `PREPEX`

  Składnia: `PREPEX address channel desired_code option`

  Przygotowuje kanał `channel` do pomiaru zewnętrznym przetwornikiem (metoda `Ctester.Prepare_External_Measurement()`).

- `PROG`

  Składnia: `PROG address channel/card file desired_code option`

  Realizuje operację programowania kontrolera kanału (jeśli argument `channel/card` ma wartość z zakresu 0 - 7) lub karty (jeśli `channel/card` przypisano 8). Plikiem użytym jako źródło nowego programu jest `file` (wymagana pełna ścieżka). Polecenie to wywołuje w celu załadowania nowego oprogramowania sekwencję funkcji: `Ctester.Chip_Enable()`, `Ctester.ADuC_Programming()` i `Ctester.Chip_Disable()`. Uwaga: programowanie nie jest wykonywane jeśli kanał został skalibrowany tj. gdy odpowiednia zmienna `tester.calibrated[]` ma wartość TRUE.

- `EXREADV`

  Składnia: `EXREADV address channel proper_result desired_code option`

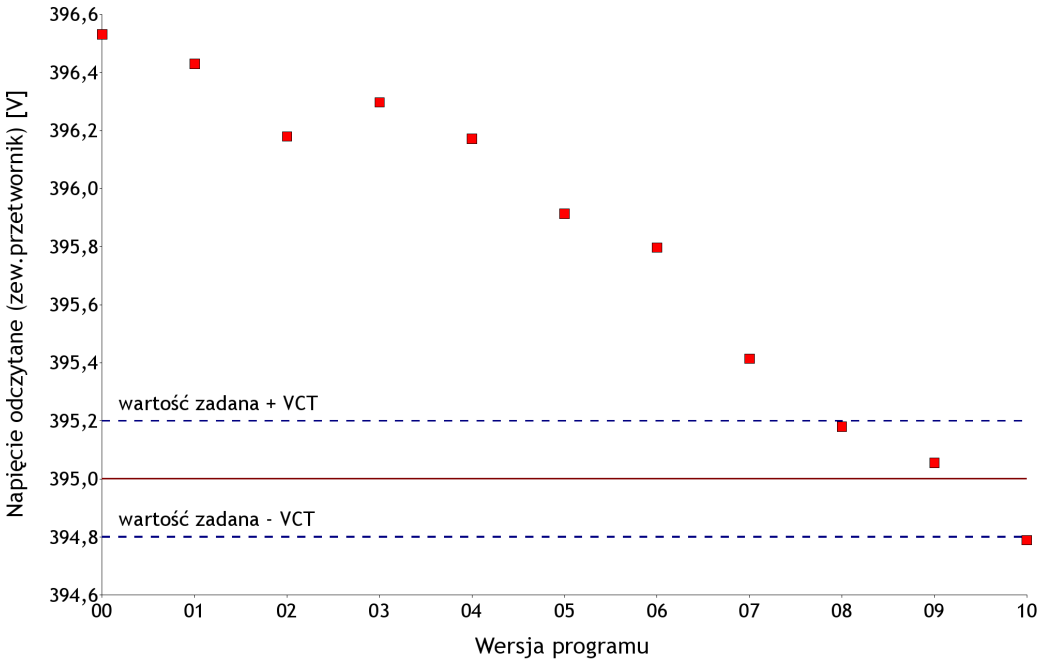
  Wykonuje odczyt napięcia kanału `channel` wykonując w tym celu metodę

`External_Read_Voltage` klasy *Ctester*. Gdy w polu opcji pojawia się słowo kluczowe `CALIBRATE` polecenie `EXREADV` uważane jest za element procedury testu kalibracji napięciowej.

Test ten polega na ładowaniu kolejnych wersji oprogramowania do mikrokontrolera wybranego kanału (rozpoczynając od wersji 0), ustawianiu żądanego napięcia i sprawdzaniu różnicy ΔU pomiędzy napięciem oczekiwanym, a napięciem zmierzonym przez przetwornik AC testera (`EXREADV`). Dąży się do tego by różnica ta była jak najmniejsza, czyli by przetwornik AC wbudowany w mikrokontroler kanału, przy użyciu którego ustawiane jest napięcie wykazywał jak najmniejszy błąd odczytu.

Teoretycznie wersja 0 programu mikrokontrolera karty powinna ustawiać napięcie rzeczywiste wyższe od napięcia zadanego, a załadowanie każdej kolejnej wersji tegoż programu (różniącą się parametrami wzmocnienia i przesunięcia charakterystyki przetwarzania AC - patrz: [11] i [12]) powinno zmniejszać napięcie odczytywane przez zewnętrzny przetwornik o ok. 0,125 V (zakładając, że wydano polecenie ustawienia napięcia 395 V które najczęściej jest w owym teście stosowane). W takim idealnym przypadku procedura kalibracji polegałaby na stopniowym zwiększaniu numeru ładowanej wersji programu i zakończeniu na pierwszym programie który wykazuje większą różnicę napięć ΔU niż jego poprzednik, a następnie załadowanie poprzedniego programu.

Niestety zależność rzeczywistego napięcia ustawianego przez mikrokontroler od wersji programu pod kontrolą którego on pracuje wykazuje pewne nieliniowości co przedstawiono na przykładowej charakterystyce (rys.4.1). Przyjmując tok rozumowania przedstawiony w poprzednim akapicie kalibracja kończyłaby się tu na załadowaniu programu o numerze 2.



Rysunek 4.1: Przykładowy przebieg kalibracji napięciowej

Dlatego też wprowadzono dodatkowy współczynnik: tolerancji kalibracji napięciowej VCT (ang. voltage calibration tolerance). Dodatkowym warunkiem zakończenia kalibracji napięciowej załadowaniem poprzedniego programu jest by różnica napięć ΔU uzyskana dla owego programu spełniała nierówność 4.1 co również zilustrowano na rys.4.1 (kalibracja kończy się na wersji 09).

$$\Delta U < |VCT| \tag{4.1}$$

Rolą polecenia `EXREADV` w trybie `CALIBRATE` jest, poza dokładnym pomiarem napięcia, sprawdzanie warunków końca kalibracji oraz załadowanie ostatecznej wersji programu mikrokontrolera kanału. Kolejne polecenia programujące nie są wykonywane (odpowiedniej zmiennej `tester.calibrated[]` nadawana jest wartość logiczna TRUE).

- `EXREADC`

  Składnia: `EXREADC address channel load desired_code option`

  Wykonuje pomiar prądu kanału `channel` przy użyciu przetwornika testera (metoda `Ctester.External_Read_Current()`) gdy obciążeniem są rezystory obciążające zdefiniowane maską `load` podaną w postaci dziesiętnej i tożsamą z bajtem LOAD ramki polecenia *external read current*.

- `MSG`

  Składnia: `MSG message`

  Zapisuje tekst `message` (nie zawierający znaków spacji!) bezpośrednio do pliku raportu (patrz: Podrozdział 4.3).

Kolejne części pracy wymieniają przykładowe testy zsyntetyzowane na bazie przedstawionych powyżej poleceń oraz prezentują postać raportów otrzymywanych w wyniku działania procedur testujących karty HV.

4.2 Przykładowe testy

W Dodatku C przedstawiono listingi kilku stworzonych przez autora skryptów testujących. Przygotowane procedury testujące obejmują:

- test prawidłowego nadawania adresu karcie HV oraz sprawdzenie czy karta ta odpowiada tylko i wyłącznie na adres przypisany jej danej chwili (plik *boardadrtest.tst*)
- test właściwego ustawiania maski (plik *masktest.tst*)
- test właściwego funkcjonowania mechanizmu interlock (plik *interlocktest.tst*)
- test prawidłowego działania zabezpieczenia napięciowego dla wybranego kanału (plik *over-voltagetest0.tst*)
- test dokładności pomiaru prądu przez przetwornik karty oraz trafnego wyboru rezystora próbnego (plik *current-calibration0.tst*
- kalibrację napięciową wybranego kanału (plik *voltage-calibration0.tst*)

4.3 Raporty

Wynik działania testu zapisywany jest w pliku tekstowym o nazwie *numer_karty.rlt* w katalogu *results*. Plik ten zawiera:

- numer testowanej karty
- nazwę pliku testu oraz ścieżkę dostępu doń
- datę i czas rozpoczęcia testu
- raport z przebiegu testu sformatowany w kolumny: numer kroku, polecenie, argumenty, oczekiwany wynik, zwrócony wynik oraz ocena poprawności uzyskanych rezultatów
- linie tekstu wstawione poleceniem `MSG`
- separatory i komentarz końcowy

Przykład pliku raportu dla kalibracji napięciowej kanału 0 karty o numerze 054 przedstawiono poniżej. Reprezentuje on proces kalibracji przedstawiony również na rys. 4.1.

```
--------------------------------------------------------------------------------
Card no: 054
Test file: C:\Program Files\Microsoft Visual Studio\MyProjects\HV_tester\Debug\
tests\voltage-calibration0.tst
Time: 06-06-2005 , 15:01.08

RESULT:
Index&Command:   Arguments:              Desired R&C:    Returned R&C:          Outcome:
  1 SBA          1                       0               0                      PASSED
----------------------
Calibrating_channel_0
----------------------
  5 WBOARD       1 255                   0               0                      PASSED
  6 WBOARD       1 0                     0               0                      PASSED
  7 WBOARD       1 1                     0               0                      PASSED
  8 PROG         1 0 chvr00.hex          0               0                      PASSED
  9 PREPEX       1 0                     0               0                      PASSED
 10 WCHANNEL     1 0 395 1000 1          0               0                      PASSED
 11 EXREADV      1 0                     395 0           396.531 0              PASSED
 12 PROG         1 0 chvr01.hex          0               0                      PASSED
 13 PREPEX       1 0                     0               0                      PASSED
 14 WCHANNEL     1 0 395 1000 1          0               0                      PASSED
 15 EXREADV      1 0                     395 0           396.430 0              PASSED
 16 PROG         1 0 chvr02.hex          0               0                      PASSED
 17 PREPEX       1 0                     0               0                      PASSED
 18 WCHANNEL     1 0 395 1000 1          0               0                      PASSED
```

```
 19 EXREADV      1 0                     395 0           396.180 0           PASSED
 20 PROG         1 0 chvr03.hex          0               0                   PASSED
 21 PREPEX       1 0                     0               0                   PASSED
 22 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 23 EXREADV      1 0                     395 0           396.297 0           PASSED
 24 PROG         1 0 chvr04.hex          0               0                   PASSED
 25 PREPEX       1 0                     0               0                   PASSED
 26 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 27 EXREADV      1 0                     395 0           396.172 0           PASSED
 28 PROG         1 0 chvr05.hex          0               0                   PASSED
 29 PREPEX       1 0                     0               0                   PASSED
 30 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 31 EXREADV      1 0                     395 0           395.914 0           PASSED
 32 PROG         1 0 chvr06.hex          0               0                   PASSED
 33 PREPEX       1 0                     0               0                   PASSED
 34 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 35 EXREADV      1 0                     395 0           395.797 0           PASSED
 36 PROG         1 0 chvr07.hex          0               0                   PASSED
 37 PREPEX       1 0                     0               0                   PASSED
 38 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 39 EXREADV      1 0                     395 0           395.414 0           PASSED
 40 PROG         1 0 chvr08.hex          0               0                   PASSED
 41 PREPEX       1 0                     0               0                   PASSED
 42 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 43 EXREADV      1 0                     395 0           395.180 0           PASSED
 44 PROG         1 0 chvr09.hex          0               0                   PASSED
 45 PREPEX       1 0                     0               0                   PASSED
 46 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 47 EXREADV      1 0                     395 0           395.055 0           PASSED
 48 PROG         1 0 chvr10.hex          0               0                   PASSED
 49 PREPEX       1 0                     0               0                   PASSED
 50 WCHANNEL     1 0 395 1000 1          0               0                   PASSED
 51 EXREADV      1 0                     395 0           394.789 0
Loaded previous program C:\ADuC\hexy\F2.9_15.02.05\chvr09.hex
CALIBRATING DONE!
144 WBOARD      1 255                   0               0                   PASSED
Done!
```

Napięcie ustawione jest najbliższe zmierzonemu zewnętrznym przetwornikiem wtedy gdy kontroler kanału korzysta z wersji programu *chvr09* i ta właśnie wersja została ostatecznie do testowanego mikrokontrolera załadowana.

Podsumowanie

Program testujący będący przedmiotem niniejszego opracowania realizuje w pełni swoje zadania. Pozwala na szybkie, automatyczne przetestowanie karty HV pod kątem wielu kryteriów wykrywając najczęściej spotykane usterki. Umożliwia również sprawną i skuteczną kalibrację napięciową która, przeprowadzana bez użycia programu *HV_tester*, jest zadaniem wyjątkowo żmudnym i czasochłonnym. Wielką korzyścią i ułatwieniem dla obsługi systemu jest także funkcja w pełni automatycznego ładowania tej wersji programu, która odpowiada przeprowadzonym pomiarom kalibracyjnym.

Ważną cechą stworzonej aplikacji jest łatwość budowy własnych procedur testujących - co wykracza poza zdefiniowany w celu pracy wymóg budowy jedynie określonego zestawu „różnorakich testów funkcjonalnych".

Programy komputerowe podlegają zwykle nieustannej ewolucji, dodawane są do nich nowe funkcje, wprowadzane poprawki itp. W przypadku opisywanej aplikacji proponowane dalsze usprawnienia to m.in.: obsługa czytnika kodów paskowych umieszczonych na każdej karcie HV i stanowiących ich identyfikator równoważny numerowi wprowadzanemu ręcznie przez użytkownika, integracja raportów w bazie danych, a także dodanie nowych poleceń rozszerzających możliwości układu testującego.

Dodatek A: Kody poleceń w układzie testującym

POLECENIE	KOD_{10}	KOD_{16}	KOD_2							
chip programming	112	0x70	0	1	1	1	0	0	0	0
external read voltage	66	0x42	0	1	0	0	0	0	1	0
external read current	17	0x11	0	0	0	1	0	0	0	1
prepare ext. measurement	58	0x3A	0	0	1	1	1	0	1	0
read board	115	0x73	0	1	1	1	0	0	1	1
read channel	102	0x66	0	1	1	0	0	1	1	0
read tester status	97	0x61	0	1	1	0	0	0	0	1
reset	113	0x71	0	1	1	1	0	0	0	1
set 5W resistors	37	0x25	0	0	1	0	0	1	0	1
set board address	103	0x67	0	1	1	0	0	1	1	1
set interlock	105	0x69	0	1	1	0	1	0	0	1
write board	100	0x64	0	1	1	0	0	1	0	0
write channel	104	0x68	0	1	1	0	1	0	0	0

Dodatek B: Kod źródłowy metod klasy *Ctester*

```
// function for reading tester's status
int Ctester::Read_Tester_Status(CSerial *tester_port, unsigned char *message)
{
        unsigned char frame[2];
        int state=4;
        int packet_length;                      // variable packet length
        struct _timeb time_start,time_stop;     // structures for time measurement
        int diff_time=0;                        // actual time spent on waiting
        int i;
        frame[0]=(unsigned char)READ_TESTER;                            // 1st byte - comm.for tester
        frame[1]=Calculate_Checksum(frame,sizeof(frame)-1);             // 2nd byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))   // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);        // waiting itself ;)
                        if(state==0 || state==8) break;                 // ACK received - break
                        if(state!=4) return state;                      // other ACK received
                }
        }
        else return 7;

        if(state==4) return state; // we didn't received ACK (of any kind in fact)
        // and now wait for data
        // first decide what are we waiting for - ADuC816 is alive ?
        if(state==0) packet_length=17;
        if(state==8) packet_length=9;

        _ftime(&time_start);
        while(1) // // infinite loop - used break & return to leave it
        {
                if(tester_port->ReadDataWaiting()>=packet_length)
                {
                        // if data waiting read it
                        tester_port->ReadData(message,packet_length); // read data packet
                        // bad checksum ?
                        if(message[packet_length-1]!=Calculate_Checksum(message,packet_length-1))
                        {
                                return 5;
                        }
                        message[packet_length-1]='\0'; // terminating string
                        break; // success - leave the loop
```

```
                }
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
                if(diff_time>TIMEOUT) return 6; // timeout passed
        }
        return 0; // successfuly read tester's status
}

//fuction for resetting card controller
int Ctester::Reset(CSerial *tester_port)
{
        unsigned char frame[2];
        int state=4;
        int i;
        frame[0]=(unsigned char)RESET;                          //1st byte - command for tester
        frame[1]=Calculate_Checksum(frame,sizeof(frame)-1);     //2nd byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))   // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);// waiting itself ;)
                        if(state!=4) return state;              // if ACK received return its code
                }
        }
        else return 7;

        return state; // we didn't received ACK (here we will return 4) ;)
}
// function for setting card address
int Ctester::Set_Board_Address(CSerial *tester_port, unsigned char address)
{
        unsigned char frame[3];
        int state=4,state_res=4;
        int i;
        frame[0]=(unsigned char)SET_BOARD; //1st byte - command for tester
        frame[1]=address;                  //2nd byte - no command for card c.(just desired address)
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1); //3rd byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))   // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state==0)
                        {
                                if(Reset(tester_port)==0)
                                {
                                        Sleep(400);
                                        card_address=address;
                                        return 0; // ACK received,and reset done - return 0
                                }
                        }
                        if(state!=4) return state;
                }
        }
        else return 7;

        return state; //failed to set board address (usually 4 here)
}

// function for writing card - setting its mask actually
```

```
int Ctester::Write_Card_HV (CSerial *tester_port, unsigned char address,unsigned char mask)
{
        unsigned char frame[4];
        int i;
        int state=4;
        frame[0]=(unsigned char)WRITE_CARD; //1st byte-command for tester
        frame[1]=address+(unsigned char)192; //2nd byte-command for card c.(SB bits set) & desired addr.
        frame[2]=mask;                      //3rd byte-mask to be set by card controller
        frame[3]=Calculate_Checksum(frame,sizeof(frame)-1);  //4th byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state!=4) return state;               // ACK - return its code
                }
        }
        else return 7;

        return state; //failed to write card (no ACK)
}

// function for reading HV card -> result in bostats buffer
int Ctester::Read_Card_HV(CSerial *tester_port, unsigned char address)
{
        unsigned char frame[3];
        unsigned char bostats[4],temp;
        int i;
        int state=4;
        struct _timeb time_start,time_stop;// structures for time measurement
        int diff_time=0;                   // actual time spent on waiting
        frame[0]=(unsigned char)READ_CARD; // 1st byte-command for tester
        frame[1]=address+(unsigned char)64;// 2nd byte-command for c. c.(only B bit set)&desired addr.
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1); //3rd byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state==0) break;                // if ACK received - break from the loop
                        if(state!=4) return state;         // ACK received - but not the right one
                }
        }
        else return 7;

        if(state==4) return state; //failed to receive ACK
        // and now wait for data
        _ftime(&time_start);
        while(1) // // infinite loop - used break & return to leave it
        {
                if(tester_port->ReadDataWaiting()>=4)
                {
                        // if data waiting read it
                        tester_port->ReadData(bostats,4); // read data packet
                        if(bostats[3]!=Calculate_Checksum(bostats,3)) return 5; // bad checksum
                        break; // success - leave the loop
                }
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
                if(diff_time>TIMEOUT) return 6; // timeout passed
```

```
        }
        // successfuly read card, now let's process some data
        //1-st byte is a status word, need further processing (bits 7,6-interlocks,5-busy)
        //interlock 4-7 (MSB)
        temp=bostats[0]&128; // mask other bits
        if(temp==128)
        {
                interlocks[1]=TRUE;
        }
        else
        {
                interlocks[1]=FALSE;
        }

        //interlock 0-3
        temp=bostats[0]&64; // mask other bits
        if(temp==64)
        {
                interlocks[0]=TRUE;
        }
        else
        {
                interlocks[0]=FALSE;
        }

        //busy
        temp=bostats[0]&32; // mask other bits
        if(temp==32)
        {
                busy=TRUE;
        }
        else
        {
                busy=FALSE;
        }
        //2nd byte is a mask
        mask=bostats[1];
        //3rd - soft.version
        software=bostats[2];
        return 0;
}

//function for reading HV channel -> result in bufinHV buffer
int Ctester::Read_Channel_HV(CSerial *tester_port, unsigned char address, unsigned char channel)
{
        unsigned char frame[4],temp;
        unsigned char bufinHV[7];
        double probe_res[4];                                    // probe resistors
        int i;
        int state=4;
        struct _timeb time_start,time_stop;     // structures for time measurement
        int diff_time=0;                                        // actual time spent on waiting
        int timeout=TIMEOUT;                    // timeout value
        probe_res[0]=62005.0;
        probe_res[1]=12005.0;
        probe_res[2]=2405.0;
        probe_res[3]=435.00;
        frame[0]=(unsigned char)READ_CHAN;      //1st byte - command for tester
        frame[1]=address;                       //2nd byte - command for card c.(SB = 0) & desired addr.
        frame[2]=channel*32;                    //3rd byte - channel to be read
```

```
        frame[3]=Calculate_Checksum(frame,sizeof(frame)-1); //4th byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);// waiting itself ;)
                        if(state==0) break;                     // ACK received - break from the loop
                        if(state!=4) return state;              // ACK received - not the right one
                }
        }
        else return 7;

        if(state==4) return state; //failed to receive ACK
        // and now wait for data
        _ftime(&time_start);
        while(1) // infinite loop - used break & return to leave it
        {
                if(tester_port->ReadDataWaiting()>=7)
                {
                        // if data waiting read it
                        tester_port->ReadData(bufinHV,7); // read data packet
                        if(bufinHV[6]!=Calculate_Checksum(bufinHV,6)) return 5; // bad checksum
                        break; // success - go out of the loop
                }
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
                if(diff_time>TIMEOUT) return 6; // timeout passed
        }
        // successfuly read channel - now some processing
        //voltage
        voltage[channel]=(bufinHV[2]*256+bufinHV[3])/8;
        //probe resistor used
        probe_res_no[channel]=bufinHV[1]&0x03;
        //current
        current[channel]=(bufinHV[4]*256+bufinHV[5])*610.5/probe_res[probe_res_no[channel]];
        temp=bufinHV[1]&0x80;
        if(temp==128) ovt[channel]=TRUE;    // over-voltage trip
                else ovt[channel]=FALSE;
        temp=bufinHV[1]&0x40;
        if(temp==64) oct[channel]=TRUE;     // over-current trip
                else oct[channel]=FALSE;
        temp=bufinHV[1]&0x10;
        if(temp==16) unst[channel]=TRUE;    // ramping in progress
                else unst[channel]=FALSE;
        temp=bufinHV[1]&0x08;
        if(temp==8) onoff[channel]=FALSE;   // off
                else onoff[channel]=TRUE;   //on
        return 0;
}

// function for external voltage measurement
int Ctester::External_Read_Voltage_HV(CSerial *tester_port, unsigned char address, unsigned char channel)
{
        unsigned char frame[2],bufinHV[3];;
        int state=4,i;
        struct _timeb time_start,time_stop; // structures for time measurement
        int diff_time=0;                    // actual time spent on waiting
        frame[0]=(unsigned char)EX_READ_VOLT; //1st byte - command for tester
        frame[1]=Calculate_Checksum(frame,sizeof(frame)-1); //2nd byte - checksum
        // and now send measure command
```

```
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);// waiting itself ;)
                        if(state==0) break;                     // if ACK received - break from the loop
                        if(state!=4) return state;              // ACK received but not the proper one
                }
        }
        else return 7;
        if(state==4) return state; //failed to receive ACK
        // and now wait for data
        _ftime(&time_start);
        while(1) // infinite loop - used break & return to leave it
        {
                if(tester_port->ReadDataWaiting()>=3)
                {
                        // if data waiting read it
                        tester_port->ReadData(bufinHV,3); // read data packet
                        if(bufinHV[2]!=Calculate_Checksum(bufinHV,2)) return 5; // bad checksum
                        break; // success - go out of the loop
                }
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
                if(diff_time>TIMEOUT) return 6; // timeout passed
        }
        // successfuly read channel - now some processing
        exact_voltage[channel]=((double)(256.0*(double)bufinHV[0]+(double)bufinHV[1]))*512.0/65536.0;
        return 0;
}

// function for external current measurement (load - 7 resistors)
int Ctester::External_Read_Current_HV(CSerial *tester_port,unsigned char address,unsigned char channel,\\
unsigned char load)
{
        unsigned char frame[3],bufinHV[4];
        double probe_res[4],probe_offset[4];     // probe resistors, offset
        int state=4,i;
        struct _timeb time_start,time_stop;      // structures for time measurement
        int diff_time=0;                         // actual time spent on waiting
        // declaring probe resistors used:
        probe_res[0]=74883.0;
        probe_res[1]=14924.0;
        probe_res[2]=2981.1;
        probe_res[3]=563.94;
        probe_offset[0]=0,66;
        probe_offset[1]=0.41;
        probe_offset[2]=0.27;
        probe_offset[3]=-2.4;
        frame[0]=(unsigned char)EX_READ_CURR; //1st byte - command for tester
        // and resistor used for measure
        frame[1]=load;
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1); //3rd byte - checksum
        // and now send measure command
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);// waiting itself ;)
                        if(state==0) break;                     // if ACK received - break from the loop
```

```
                    if(state!=4) return state;              // ACK received but not the proper one
                }
        }
        else return 7;
        if(state==4) return state; //failed to receive ACK
        // and now wait for data
        _ftime(&time_start);
        while(1) // infinite loop - used break & return to leave it
        {
                if(tester_port->ReadDataWaiting()>=4)
                {
                        // if data waiting read it
                        tester_port->ReadData(bufinHV,4); // read data packet
                        if(bufinHV[3]!=Calculate_Checksum(bufinHV,3)) return 5; // bad checksum
                        break; // success - go out of the loop
                }
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
                if(diff_time>TIMEOUT*10) return 6; // timeout passed
        }
        // successfuly read channel - now some processing
        test_res=bufinHV[2]&0x03;
        //current
        exact_current[channel]=(double)((double)bufinHV[0]*256+(double)bufinHV[1])\\
        *62.5/probe_res[test_res]-probe_offset[test_res];
        return 0;
}

// prepare for external measurement - should be done before each series of ext. measurement
// and before writing to measured channel (setting voltage)
// note: after programming you need to prepare again
int Ctester::Prepare_External_Measurement(CSerial *tester_port, unsigned char address,\\
unsigned char channel)
{
        unsigned char frame[3],maskoff;
        int state,state2,i;
        maskoff=(unsigned char)(pow(2,channel));
        frame[0]=(unsigned char)PREPARE_EX_READ; //1st byte - command for tester
        frame[1]=maskoff; //second byte is the mask of measured channel
        maskoff=~maskoff; // now the mask should be inverted (0 where we want to measure)
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1); //3rd byte - checksum
        // first maskoff selected channel
        state=Read_Card_HV(tester_port,address);
        if(state!=0) return state;
        state=Write_Card_HV(tester_port,address,mask&maskoff);
        Sleep(100);     // just for safety precautions
        if(state!=0) return state;
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))   // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state==0)            // if ACK received - turn on the channel
                        {
                                state2=Write_Card_HV(tester_port,address,mask|frame[1]);
                                Sleep(100);     // just for safety precautions
                                return state2;
                        }
                }
        }
```

```
        else return 7;
        return state;
}

int Ctester::Set_5W_Resistors(CSerial *tester_port,unsigned char mask_5W)
{
        int state=4;
        unsigned char frame[3];
        int i;
        frame[0]=(unsigned char)SET_5WR;                    //1st byte - command for tester
        frame[1]=mask_5W;                                   //2nd byte - mask of resistors
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1);
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);// waiting itself ;)
                        if(state!=4) return state;                  // ACK received - return its code
                }
        }
        else return 7;
        return state; //failed to write channel (no ACK)
}

int Ctester::Write_Channel_HV(CSerial *tester_port, unsigned char address, unsigned char channel,\\
double setHV[3])
{
        //this will be a real mess ;)
        int state=4;
        unsigned char frame[18];
        double probe_res[4];                        // probe resistors
        probe_res[0]=62005.0;
        probe_res[1]=12005.0;
        probe_res[2]=2405.0;
        probe_res[3]=435.00;
        int voltage,current;                        //voltage,current (int levels)
        int i;
        frame[0]=(unsigned char)WRITE_CHAN;         //1st byte - command for tester
        frame[1]=address+(unsigned char)128;        //2nd byte - command for card c.(S=1,B=0)&desired addr.
        frame[2]=channel*32;                        //3rd byte - channel to be written
        frame[3]=12;                                //4th byte - no of bytes for channel controller
        frame[4]=32;                                //5th byte - command code - SET
        //calculate voltage part
        voltage=(int)(setHV[0]*8.0);                //first - convert to int level
        frame[5]=voltage/256;                       //6th byte - 4 MSBits of voltage
        frame[6]=voltage%256;                       //7th byte - 8 LSBits of voltage
        //calculate trip limits (for each probe resistor)
        for(i=0;i<4;i++)
        {
                current=(int)(setHV[1]*probe_res[i] * 0.001638);
                frame[7+(2*i)]=current/256;
                frame[7+(2*i+1)]=current%256;
                if(frame[7+(2*i)]>15)
                {
                        frame[7+(2*i)]=0x0F;
                        frame[7+(2*i+1)]=0xFF;
                }
                //bytes 8-15 contain trip limits (HIGH,LOW) for each probe resistor
        }
        //16th byte - ramping code (0->no ramping,1->50V/s,2->20V/s,3->10V/s,4->5V/s)
```

```
        frame[15]=(unsigned char)setHV[2];
        frame[16]=0xEC;                                 //17th byte - marker
        frame[17]=Calculate_Checksum(frame,sizeof(frame)-1);
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state!=4) return state;                // if ACK received return its code
                }
        }
        else return 7;
        return state; //failed to write channel (no ACK)

}

int Ctester::Set_Interlock(CSerial *tester_port, unsigned char address, unsigned char locks)
{
        unsigned char frame[3];
        int state=4;
        int i;
        frame[0]=(unsigned char)SET_INTERLOCK;                //1st byte - command for tester
        frame[1]=locks*64;                                    //2nd byte - interlocks to be set (MSBits)
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1); //3rd byte - checksum
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT);  // waiting itself ;)
                        if(state!=4) return state;                // ACK received - return its code
                }
        }
        else return 7;
        return state;
}

int Ctester::Chip_Enable(CSerial *tester_port,unsigned char address,unsigned char channel)
{
        unsigned char frame[3];
        int i,state_int;
        int state=4;
        frame[0]=SET_PROG;
        if(channel==8) frame[1]=2;
        else frame[1]=1;
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1);
        //first clear the mask
        state_int=Write_Card_HV(tester_port,address,(unsigned char)0);
        if(state_int!=0) return state_int;
        Sleep(2000); // needed to be sure that all channels are off
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        if(state==0)
                        // if ACK received - set mask on selected channel
                        {
                                if(channel!=8)
                                {
                                        // programming channel controller
```

```
                                        state_int=Write_Card_HV(tester_port,address,(unsigned char)\\
                                        (pow(2,channel)));
                                        if(state_int==0)
                                        {
                                                Sleep(500); // wait for this channel to turn on
                                                return 0;
                                        }
                                        else return state_int;
                                }
                                else
                                {
                                        // programming card controller
                                        // here we should reset
                                        state_int=Reset(tester_port);
                                        if(state_int==0)
                                        {
                                                return 0;
                                                // wait for card controller to "get up" in loader mode
                                                Sleep(500);
                                        }
                                        else return state_int;
                                }
                        }
                        else return state;
                }
                return state; // failed to enable programming mode
        }
        else return 7;
}


int Ctester::Chip_Disable(CSerial *tester_port,unsigned char channel)
{
        unsigned char frame[3];
        int i,state_two;
        int state=4;
        frame[0]=SET_PROG;
        frame[1]=0;
        frame[2]=Calculate_Checksum(frame,sizeof(frame)-1);
        if(tester_port->SendData(frame,sizeof(frame))==sizeof(frame))    // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(tester_port,TIMEOUT); // waiting itself ;)
                        // programming card controller - reset it
                        if(state==0 && channel==8)
                        {
                                state_two=Reset(tester_port);  // if ACK received - reset card
                                Sleep(300);
                                return state_two;
                        }
                        // programming channel controller - turn off and then turn on
                        if(state==0 && channel!=8)
                        {
                                state_two=Write_Card_HV(tester_port,card_address,(unsigned char)0);
                                if (state_two==0)
                                {
                                        state_two=Write_Card_HV(tester_port,card_address,\\
                                        (unsigned char)pow(2,channel));
                                        Sleep(300);
```

```
                                        return state_two;
                                }
                        }
                }
        }
        else return 7;

        return state; // if state != 0 then return it
}

int Ctester::ADuC_Programming(CSerial *prog_port,CString hex_file_name)
{
        BOOL done=FALSE;
        CStdioFile hex_file;
        CString strLine;
        unsigned char frame[128],sum;
        int bytecount,record_type,state,i,packetsum,iteration=0;
        unsigned long addr;
        unsigned char *data;
        // unsigned char rec_buffer[25];          // for interrogating phase
        // struct _timeb time_start,time_stop;    // structures for time measurement (interrogating phase)
        int diff_time=0;                          // actual time spent on waiting
        // first let's open hex file
        if(!(hex_file.Open(hex_file_name, CFile::modeRead | CFile::typeText)))
                return 11;
        // then ask the loader if it's alive (reset it)
        Sleep(500);
        frame[0]=0x21;
        frame[1]=0x5A;
        frame[2]=0x00;
        frame[3]=0xA6;
        prog_port->SendData(frame,1);
        Sleep(500);

        // interrogating phase - for purists
        /* while(iteration<RETRIES && done==FALSE)
        {
        prog_port->SendData(frame,1);
        Sleep(400);
        if(prog_port->SendData(&frame[1],3)==3) // first check if data sent
        {
                _ftime(&time_start);              // start the timer
                while(1)                          // infinite loop - used break & return to leave it
                {
                        if(prog_port->ReadDataWaiting()>=25)
                        {
                                // if data waiting read it
                                prog_port->ReadData(rec_buffer,25); // read data packet
                                // check for checksum here (second complement)
                                if(rec_buffer[24]==(unsigned char)(0-Calculate_Checksum(rec_buffer,24)))
                                {
                                        done=TRUE;
                                        break;                    // success - go out of the loop
                                }
                                else
                                {
                                        state=5;                  // failed checksum test
                                        break;                    // try again
                                }
                        }
```

```
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm\\
                -time_start.millitm);
                if(diff_time>TIMEOUT)
                {
                        state=6;                        // timeout passed
                        break;                          // try again
                }
        }
}
else return 7;
Sleep(200);
iteration++;
}
if(done==FALSE) return state; */
//end of interrogating

done=FALSE;
iteration=0;
// clear flash/EE program and data memory
frame[0]=0x07;
frame[1]=0x0E;
frame[2]=0x01;
frame[3]=0x41;
frame[4]=0xBE;

while(iteration<RETRIES && done==FALSE)
{
        if(prog_port->SendData(frame,5)==5)     // first check if data sent
        {
                for(i=0;i<RETRIES;i++) // waiting for ACK repeated for desired no of retries
                {
                        state=Wait_for_ACK(prog_port,TIMEOUT,TRUE); // waiting itself ;)
                        if(state==0)
                        {
                                done=TRUE;
                                break;
                        }
                }
        }
        else return 7;
        iteration++;
        Sleep(300);             // between retries let the ADuC to rest a while
}
if(done==FALSE) return state;

// and now - read the record in the hex-file, send it, wait for ACK (iteratively)
while (hex_file.ReadString(strLine) != NULL)
{
        sum=0;
        if(strLine[0]!=':') // check for begining character
        {
                hex_file.Close();
                return 12; // not an Intel hex-file
        }
        // read record length
        if(sscanf(strLine.GetBuffer(strLine.GetLength())+1,"%2X",&bytecount)!=1)
        {
                hex_file.Close();
                return 12;
```

```
}
sum+=bytecount;
// read load address
if(sscanf(strLine.GetBuffer(strLine.GetLength())+3,"%4lX",&addr)!=1)
{
        hex_file.Close();
        return 12;
}
sum+=(unsigned char)((addr)/256);
sum+=(unsigned char)((addr)%256);
// read record type
if(sscanf(strLine.GetBuffer(strLine.GetLength())+7,"%2X",&record_type)!=1)
{
        hex_file.Close();
        return 12;
}
if (record_type==1)
{
        hex_file.Close();
        return 0;       // end of hex - file return succesful code
}
if (record_type!=0)
{
        hex_file.Close();
        return 12;      // unknown record type
}
sum+=record_type;
data=new unsigned char[bytecount+8];    // allocate memory for the packet
data[0]=0x07;                           // form loader packet
data[1]=0x0E;
data[2]=bytecount+4;
data[3]='W';
data[4]= (unsigned char)((addr>>16)%256);
data[5]= (unsigned char)((addr>> 8)%256);
data[6]= (unsigned char)(addr % 256);
// and now read data bytes
for(i=0;i<bytecount;i++)
{
        if(sscanf(strLine.GetBuffer(strLine.GetLength())+9+i*2,"%2X",&data[7+i])!=1)
        {
        hex_file.Close();
        return 12;      // not a hex file ?
        }
        sum+=data[7+i];
}
// and now read the checksum
if(sscanf(strLine.GetBuffer(strLine.GetLength())+9+bytecount*2,"%2X",&packetsum)!=1)
{
        hex_file.Close();
        return 12;      // not a hex file ?
}
sum+=packetsum;
// and validate it
if(sum!=0)
{
        hex_file.Close();
        return 5;
}
//write the checksum to the end of the packet
sum=bytecount+4;
```

```
                for(i=0;i<(bytecount+4);i++)
                sum+=data[i+3];
                data[bytecount+7]=0-sum;
                // and now send the packet :)
                for(i=0;i<RETRIES;i++) // sending repeated for desired no of retries
                {
                        // first check if data sent
                        if(prog_port->SendData(data,bytecount+8)==bytecount+8)
                        {
                                // waiting for ACK repeated for desired no of retries
                                for(i=0;i<RETRIES;i++)
                                {
                                        state=Wait_for_ACK(prog_port,TIMEOUT,TRUE);
                                        // waiting itself ;)
                                        if(state==0) break;
                                }
                        }
                        else state=7;
                        // if state==0 go out of this loop too
                        if(state==0) break;
                }
                if(state!=0) return state;
        }
        hex_file.Close();
        return 0;
}


// Wait for acknowledge from the tester
// Note: timeout < 1000ms !
int Ctester::Wait_for_ACK(CSerial *tester_port,int timeout,BOOL programming)
{
        struct _timeb time_start,time_stop; // structures for time measurement
        unsigned char test_byte=0;          // byte received (should be ACK when ACK;)
        int diff_time=0;                    // actual time spent on waiting
        _ftime(&time_start);
        while(test_byte!=(unsigned char)ACK && diff_time<timeout)
        {
                // first check for incoming data
                if(tester_port->ReadDataWaiting()>0)
                {
                        // if data waiting read and check if it's ACK
                        tester_port->ReadData(&test_byte,1);
                        // here we test for received byte (which ACK)
                        switch(test_byte)
                        {
                        case ACK_ADUC_DEAD:     // received ACK_ADUC_DEAD ERROR
                                return 8;
                                break;
                        case ACK_CRC_ERROR:
                                return 1;       // received ACK CRC ERROR
                                break;
                        case ACK_BUSY_ERROR:
                                return 2;       // received ACK BUSY ERROR
                                break;
                        case ACK_CARD_DEAD:
                                return 3;       // received ACK CARD DEAD
                                break;
                        case ACK:
                                if(programming==FALSE) return 0; // received ACK - OK!
```

```
                        break;
                    case NACK_PROG:
                        if(programming==TRUE) return 13; // received NACK
                        break;
                    case ACK_PROG:
                        if(programming==TRUE) return 0;  // received ACK - OK!
                        break;
                    default:
                        break;// received something weird - continue to wait
                    }
                }
                // else : measure time
                _ftime(&time_stop);
                diff_time=((time_stop.time-time_start.time)*1000+time_stop.millitm-time_start.millitm);
        }
        return 4; // failed to receive any ACK
}

unsigned char Ctester::Calculate_Checksum(unsigned char *data,int length)
{
        unsigned char sum=0;
        int i;
        for(i=0;i<length;i++) sum+=data[i];
        return sum;
}
```

Dodatek C: Skrypty testujące

1. boardadrtest.tst

```
HV_TESTER_TEST_FILE
Proper address setting test
# first we set board address
SBA 0 0 BREAK
# set mask - just to make sure
WBOARD 0 0 0 BREAK
# check if card responds at selected address
RBOARD 0 0 0 BREAK
# then check if card responds to bad ones
RBOARD 1 0 3 BREAK
RBOARD 2 0 3 BREAK
RBOARD 3 0 3 BREAK
RBOARD 4 0 3 BREAK
RBOARD 5 0 3 BREAK
RBOARD 6 0 3 BREAK
RBOARD 7 0 3 BREAK
RBOARD 8 0 3 BREAK
RBOARD 9 0 3 BREAK
RBOARD 10 0 3 BREAK
RBOARD 11 0 3 BREAK
RBOARD 12 0 3 BREAK
RBOARD 13 0 3 BREAK
RBOARD 14 0 3 BREAK
RBOARD 15 0 3 BREAK
RBOARD 16 0 3 BREAK
RBOARD 17 0 3 BREAK
RBOARD 18 0 3 BREAK
RBOARD 19 0 3 BREAK
RBOARD 20 0 3 BREAK
RBOARD 21 0 3 BREAK
RBOARD 22 0 3 BREAK
RBOARD 23 0 3 BREAK
RBOARD 24 0 3 BREAK
RBOARD 25 0 3 BREAK
RBOARD 26 0 3 BREAK
RBOARD 27 0 3 BREAK
RBOARD 28 0 3 BREAK
RBOARD 29 0 3 BREAK
RBOARD 30 0 3 BREAK
RBOARD 31 0 3 BREAK
# next we set new board address (31)
SBA 31 0 BREAK
# set mask - just to make sure
WBOARD 31 0 0 BREAK
```

```
# check if card responds at selected address
RBOARD 31 0 0 BREAK
# then check if card responds to bad ones
RBOARD 0 0 3 BREAK
RBOARD 1 0 3 BREAK
RBOARD 2 0 3 BREAK
RBOARD 3 0 3 BREAK
RBOARD 4 0 3 BREAK
RBOARD 5 0 3 BREAK
RBOARD 6 0 3 BREAK
RBOARD 7 0 3 BREAK
RBOARD 8 0 3 BREAK
RBOARD 9 0 3 BREAK
RBOARD 10 0 3 BREAK
RBOARD 11 0 3 BREAK
RBOARD 12 0 3 BREAK
RBOARD 13 0 3 BREAK
RBOARD 14 0 3 BREAK
RBOARD 15 0 3 BREAK
RBOARD 16 0 3 BREAK
RBOARD 17 0 3 BREAK
RBOARD 18 0 3 BREAK
RBOARD 19 0 3 BREAK
RBOARD 20 0 3 BREAK
RBOARD 21 0 3 BREAK
RBOARD 22 0 3 BREAK
RBOARD 23 0 3 BREAK
RBOARD 24 0 3 BREAK
RBOARD 25 0 3 BREAK
RBOARD 26 0 3 BREAK
RBOARD 27 0 3 BREAK
RBOARD 28 0 3 BREAK
RBOARD 29 0 3 BREAK
RBOARD 30 0 3 BREAK
```

2. `masktest.tst`

```
HV_TESTER_TEST_FILE
Proper mask setting test
# first we set board address
# SBA address desired_code option
SBA 1 0 BREAK
# then do the mask setting
WBOARD 1 0 0 BREAK
# and check it
RBOARD 1 0 0 BREAK
# other masks
WBOARD 1 1 0 BREAK
RBOARD 1 1 0 BREAK
WBOARD 1 2 0 BREAK
RBOARD 1 2 0 BREAK
WBOARD 1 4 0 BREAK
RBOARD 1 4 0 BREAK
WBOARD 1 8 0 BREAK
RBOARD 1 8 0 BREAK
WBOARD 1 16 0 BREAK
RBOARD 1 16 0 BREAK
WBOARD 1 32 0 BREAK
RBOARD 1 32 0 BREAK
WBOARD 1 64 0 BREAK
```

```
RBOARD 1 64 0 BREAK
WBOARD 1 128 0 BREAK
RBOARD 1 128 0 BREAK
WBOARD 1 0 0 BREAK
```

3. interlocktest.tst

```
HV_TESTER_TEST_FILE
Proper interlock setting test
# first we set board address
SBA 1 0 BREAK
# then do the mask setting
WBOARD 1 255 0 BREAK
# make sure it's ok
RBOARD 1 255 0 BREAK
# set interlock 0-3
WINTERLOCK 1 1 0 BREAK
# check the mask (should be 240)
RBOARD 1 240 0 BREAK
# check if interlock read well
RINTERLOCK 1 1 0 BREAK
# restore
WINTERLOCK 1 0 0 BREAK
WBOARD 1 255 0 BREAK
# now mask should be 255
RBOARD 1 255 0 BREAK
# and interlocks 0
RINTERLOCK 1 0 0 BREAK
# set interlock 4-7
WINTERLOCK 1 2 0 BREAK
# check the mask (should be 15)
RBOARD 1 15 BREAK
# check if interlock read well
RINTERLOCK 1 2 0 BREAK
# restore
WINTERLOCK 1 0 0 BREAK
WBOARD 1 255 0 BREAK
# now mask should be 255
RBOARD 1 255 0 BREAK
# and interlock 0:
RINTERLOCK 1 0 0 BREAK
# go back to initial state
WBOARD 1 0 0 BREAK
RBOARD 1 0 0 BREAK
```

4. over-voltage0.tst

```
HV_TESTER_TEST_FILE
Proper over-voltage functionality test for channel 0
# first we set board address
SBA 0 0 BREAK
WBOARD 0 255 0 BREAK
WBOARD 0 0 0 BREAK
MSG ------------------
MSG Testing_channel_0
MSG ------------------
WBOARD 0 1 0 BREAK
# set voltage
```

```
WCHANNEL 0 0 470 1000 1 0 BREAK
# check if over-voltage ocurred
ROVT 0 0 0 0
WCHANNEL 0 0 471 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 472 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 473 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 474 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 475 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 476 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 477 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 478 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 479 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 480 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 481 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 482 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 483 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 484 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 485 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 486 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 487 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 488 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 489 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 490 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 491 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 492 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 493 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 494 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 495 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 496 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 497 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 498 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 499 1000 1 0 BREAK
```

```
ROVT 0 0 0 0
WCHANNEL 0 0 500 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 501 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 502 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 503 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 504 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 505 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 506 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 507 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 508 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 510 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 511 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 512 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 513 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 514 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 515 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 516 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 517 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 518 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 519 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 520 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 521 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 522 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 523 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 524 1000 1 0 BREAK
ROVT 0 0 0 0
WCHANNEL 0 0 525 1000 1 0 BREAK
ROVT 0 0 0 0
WBOARD 0 255 0 BREAK
MSG ------------------
MSG End_of_test
MSG ------------------
```

5. current-calibration0.tst

```
HV_TESTER_TEST_FILE
Current calibration (ch0)-compares current measure for different loads
```

```
# OK SO LET'S_START
# first we set board address
SBA 0 0 BREAK
# set mask - channel 0
WBOARD 0 0 0 BREAK
WBOARD 0 1 0 BREAK
# and now prepare for measurement
PREPEX 0 0 0 BREAK
# set voltage
MSG ------------------------------
MSG LOAD=1129650K_VOLTAGE=17V
MSG ------------------------------
WCHANNEL 0 0 17 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 254 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 0 0
MSG ------------------------------
MSG LOAD=9650K_VOLTAGE=32V
MSG ------------------------------
WCHANNEL 0 0 32 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 30 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 0 0
MSG ------------------------------
MSG LOAD=9650K_VOLTAGE=43V
MSG ------------------------------
WCHANNEL 0 0 43 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 30 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 1 0
MSG ------------------------------
MSG LOAD=1050K_VOLTAGE=200V
MSG ------------------------------
WCHANNEL 0 0 200 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 14 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 1 0
MSG ------------------------------
MSG LOAD=1050K_VOLTAGE=231V
MSG ------------------------------
WCHANNEL 0 0 231 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 14 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 2 0
MSG ------------------------------
```

```
MSG LOAD=110K_VOLTAGE=105V
MSG ----------------------------
WCHANNEL 0 0 105 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 6 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 2 0
MSG ----------------------------
MSG LOAD=110K_VOLTAGE=121V
MSG ----------------------------
WCHANNEL 0 0 121 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 6 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
RPROBE 0 0 3 0
MSG ----------------------------
MSG LOAD=10K_VOLTAGE=49V
MSG ----------------------------
WCHANNEL 0 0 49 5000 1 0 BREAK
# measure current using external converter
EXREADC 0 0 2 0 BREAK
# measure current using internal converter
RCURVOLT 0 0 0 BREAK
# check probe used
WBOARD 0 0 0 BREAK
MSG ------------
MSG END_OF_TEST
MSG ------------
```

6. voltage-calibration0.tst

```
HV_TESTER_TEST_FILE
Voltage calibration - loads suitable program ver. into channel 0 controller
# SBA address desired_code option
SBA 1 0 BREAK
MSG ----------------------
MSG Calibrating_channel_0
MSG ----------------------
WBOARD 1 255 0 BREAK
WBOARD 1 0 0 BREAK
WBOARD 1 1 0 BREAK
# load the program 00
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr00.hex 0 BREAK
# prepare to measure
PREPEX 1 0 0 BREAK
# set voltage
WCHANNEL 1 0 395 1000 1 0 BREAK
# and measure - with calibration
EXREADV 1 0 395 0 CALIBRATE BREAK
# continue to program
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr01.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr02.hex 0 BREAK
PREPEX 1 0 0 BREAK
```

```
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr03.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr04.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr05.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr06.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr07.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr08.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr09.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr10.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr11.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr12.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr13.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr14.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr15.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr16.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr17.hex 0 BREAK
PREPEX 1 0 0 BREAK
```

```
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr18.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr19.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr20.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr21.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr22.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr23.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr24.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr25.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr26.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr27.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr28.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr29.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr30.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr31.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr32.hex 0 BREAK
PREPEX 1 0 0 BREAK
```

```
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
PROG 1 0 C:\ADuC\hexy\F2.9_15.02.05\chvr33.hex 0 BREAK
PREPEX 1 0 0 BREAK
WCHANNEL 1 0 395 1000 1 0 BREAK
EXREADV 1 0 395 0 CALIBRATE BREAK
WBOARD 1 255 0 BREAK
WBOARD 1 0 0 BREAK
MSG ------------------
MSG End_of_test
MSG ------------------
```

Dodatek D: Zawartość płyty CD

Do opracowania dołączono nośnik zawierający zarówno elektroniczną wersję niniejszego dokumentu w formacie LaTeX (katalog *documentation*) jak i kod źródłowy wykonanego programu (katalog *source*). Jego funkcjonalność została pozytywnie zweryfikowana na komputerze klasy PC wyposażonym w system operacyjny Windows Server 2003. Do kompilacji użyto licencjonowanej wersji środowiska Microsoft Visual Studio w wersji 6.0.

Bibliografia

[1] Z. Cyganek, „Projekt i budowa testera dla wielokanałowych zasilaczy detektorów krzemowych, sterowanych mikrokontrolerami"-praca magisterska, niepublikowana, 2005

[2] E. Górnicki, S. Koperny, P. Malecki, E. Stanecka, „SCT High Voltage Power Supply: Requirements and Specifications, ver. 3.06", ATLAS Project document no. ATL-IS-ES-0084, 2004

[3] D.E. Comer, „Sieci komputerowe i intersieci", Wydawnictwo Naukowo-Techniczne, 2001

[4] P. Metzger, „Anatomia PC", Wydawnictwo Helion, 2004

[5] A. Daniluk, „RS 232C Praktyczne programowanie", Wydawnictwo Helion, 2001

[6] Analog Devices, „Understanding the Serial Download Protocol", Microconverter Technical Note uC004, 2001

[7] D. Chapman, „Teach Yourself Visual C++ 6 in 21 days", Sams Publishing, 1998

[8] R.C Leinecker, T. Archer, „Visual C++ 6: Vademecum Profesjonalisty", Wydawnictwo Helion, 2000

[9] C.H. Pappas, W.H. Murray III, „Visual C++ 6: Kompendium Wiedzy", Wydawnictwo PLJ, 2000

[10] R. Bhavnani, „A serialization primer", The Code Project (www.codeproject.com), 2002

[11] Analog Devices, „ADμ C812 MicroConverter, Multichannel 12-Bit ADC with Embedded Flash MCU", Microconverter Data Sheet rev. E, 2003

[12] Analog Devices, „ADuC812 ADC Software Calibration", Microconverter Technical Note uC005, 2001

Spis tabel

Spis rysunków